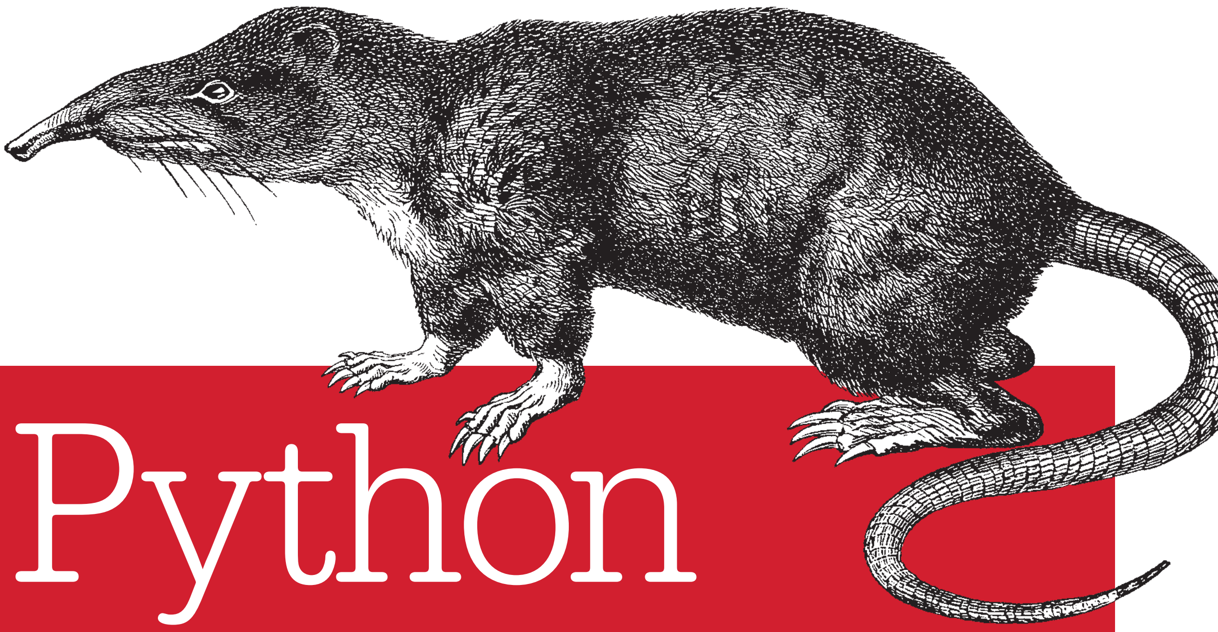
O'REILLY®

异步图书 www.epubit.com.cn

第2版



Python 金融大数据分析

Python for Finance, Second Edition

[德] 伊夫·希尔皮斯科（Yves Hilpisch）著

姚军 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

（第 2 版）

Python 金融大数据分析

[德]伊夫·希尔皮斯科（Yves Hilpisch） 著

姚 军 译

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

Python金融大数据分析 ：第2版 / （德）伊夫·希尔皮斯科著 ；姚军译. -- 北京 ：人民邮电出版社，2020.4
ISBN 978-7-115-52133-0

Ⅰ. ①P… Ⅱ. ①伊… ②姚… Ⅲ. ①软件工具—程序设计—应用—金融—分析 Ⅳ. ①F830.41-39

中国版本图书馆CIP数据核字(2019)第211495号

◆ 著　　　　[德] 伊夫·希尔皮斯科（Yves Hilpisch）
译　　　　姚　军
责任编辑　武晓燕
责任印制　王　郁　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：42
字数：871 千字　　　　2020 年 4 月第 1 版
印数：1 – 3 000 册　　　　2020 年 4 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-8711 号

定价：139.00 元

读者服务热线：(010) 81055410　印装质量热线：(010) 81055316
反盗版热线：(010) 81055315
广告经营许可证：京东工商广登字 20170147 号

内容提要

本书分为 5 部分，共 21 章。第 1 部分介绍了 Python 在金融学中的应用，其内容涵盖了 Python 用于金融行业的原因、Python 的基础架构和工具，以及 Python 在计量金融学中的一些具体入门实例；第 2 部分介绍了 Python 的基础知识以及 Python 中非常有名的库 NumPy 和 pandas 工具集，还介绍了面向对象编程；第 3 部分介绍金融数据科学的相关基本技术和方法，包括数据可视化、输入/输出操作和数学中与金融相关的知识等；第 4 部分介绍 Python 在算法交易上的应用，重点介绍常见算法，包括机器学习、深度神经网络等人工智能相关算法；第 5 部分讲解基于蒙特卡洛模拟开发期权及衍生品定价的应用，其内容涵盖了估值框架的介绍、金融模型的模拟、衍生品的估值、投资组合的估值等知识。

本书适合对使用 Python 进行大数据分析和处理感兴趣的金融行业开发人员阅读。

O’Reilly Media, Inc.介绍

O’Reilly以“分享创新知识、改变世界”为己任。40多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来O’Reilly图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

> “O’Reilly Radar博客有口皆碑。”
>
> ——Wired

> “O’Reilly凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”
>
> ——Business 2.0

> “O’Reilly Conference是聚集关键思想领袖的绝对典范。”
>
> ——CRN

> “一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”
>
> ——Irish Times

> “Tim是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”
>
> ——Linux Journal

前言

近来，Python 无疑是金融业的重要策略性技术平台之一。当我于 2013 年开始编写本书第 1 版时，仍然在许多演讲中不懈地介绍 Python 在金融方面相比其他语言及平台的竞争优势。到 2018 年年底，这已经不再是个问题：全世界的金融机构现在都尽最大努力利用 Python 及其强大的数据分析、可视化和机器学习程序库生态系统。在金融领域之外，Python 还常常成为编程入门课程（例如计算机科学课程项目）选择的语言。

除了容易理解的语法和多重范型方法之外，形成这一局面的主要原因之一是，Python 已经成为人工智能（AI）、机器学习（ML）和深度学习（DL）领域的“头等公民”。这些领域的许多流行的软件包和程序库都直接用 Python（如 ML 所用的 scikit-learn）编写，或者用 Python 包装器（例如 DL 所用的 TensorFlow）。

金融本身正在进入一个新时代，这一发展有两个主要的推动力。首先是基本上所有金融数据都可以编程访问——一般来说，这种访问是实时的，也是催生“数据驱动金融”的原因。几十年前，大部分交易或者投资决策是由交易员和投资组合管理人推动的，这些人阅读报纸，或者通过私下交谈进行学习。此后出现了一些终端，可通过计算机和电子通信，将金融数据实时传递到交易员和投资组合管理人的桌面上。今天，即使是一分钟产生的海量金融数据，个人（或者团队）都无法应付。只有处理速度和计算能力与日俱增的机器，才能应对金融数据的容量和速度。这意味着，全球大部分股票交易量是由算法和计算机驱动的，而不是交易员。

第二个主要推动力是在金融中越来越重要的人工智能。越来越多的金融机构试图利用 ML 和 DL 算法改善运营，以及它们的交易及投资表现。2018 年初，第一本关于“金融机器学习”的专著出版，强调了这种趋势。毫无疑问，还会有更多的图书涌现。这导致了可以称为“人工智能优先金融学”的现象，用可参数化的 ML 及 DL 算法代替传统的金融理论。传统的金融理论可能非常优雅，但在数据驱动、人工智能优先的金融学实践中不再有太大的用处了。

面对这一金融时代的挑战，Python 是合适的编程语言和生态系统。本书涵盖有监督学习和无监督学习的基本 ML 算法（以及深度神经网络），但焦点是 Python 的数据处理和分析能力。想要全面叙述 AI 当前和未来在金融中的重要性，需要一整本书的篇幅。不过，大部分 AI、ML 和 DL 需要大量的数据，因此无论如何应该首先掌握数据驱动金融学。

本书第 2 版更多的是一次升级，而非更新。例如，这一版增加了关于算法交易的一整个部分（第 4 部分）。这一主题最近在金融业变得相当重要，在散户中也很受欢迎。本

版还增加了一个入门部分（第 2 部分），介绍 Python 基本编程和数据分析，这些知识将为本书的后续几个部分奠定基础。另一方面，第 1 版的某些章节完全删除了。例如，关于 Web 技术和对应库（如 Flask）的部分删除，因为现在已经有专门介绍这些知识的图书。

在第 2 版中，我力图涵盖更多金融相关主题，聚焦于对金融数据科学、算法交易和计算金融学特别有用的 Python 技术。和第 1 版中一样，我采用的是实用方法，实现和图示先于理论细节，通常将重点放在整体上，而不是某些类、方法或者晦涩难懂的函数参数化选项。

描述完第 2 版的基本方法之后，我还必须强调，这本书既不是介绍 Python 编程，也不是介绍一般金融知识的图书。在这两个方面，都有大量出色的知识来源。本书定位于这两个激动人心的领域相互交叉的方面，并假定读者有一定的编程（不一定是 Python）和金融背景。这些读者将学习把 Python 及其生态系统应用于金融领域的方法。

Jupyter Notebook 和本书配套代码可以通过 Quants 平台访问和执行。读者可以在 Python Quants 官网免费注册。

我的公司（Python Quants）和本人还提供了许多帮助读者掌握 Python 金融数据科学、人工智能、算法交易和计算金融学的资源。你可以从访问如下站点开始：

- 我们公司的网站 Python Quants；
- 我的个人网站 The AI QUANT；
- 我们的 Python 书籍网站（在 Python Quants 官网中）；
- 我们的在线培训网站（在 Python Quants 官网中）；
- 我们的认证计划网站（在 Python Quants 官网中）。

在我们前几年所提供的服务中，最令我自豪的是“Python 算法交易认证计划”（*Certificate Program in Python for Algorithmic Trading*）。它提供了超过 150 小时的实时和录制教程，超过 1200 页的文档，5000 行以上的 Python 代码以及 50 多个 Jupyter Notebook。这一课程每年开设多次，每一次我们都进行更新和改进。这些在线课程是同类中的首创。通过与萨尔州应用技术大学的合作，成功完成该课程者将获得正式的大学文凭。

除此之外，我最近还启动了新的“The AI Machine”项目和公司，以实现自动化算法交易策略部署的标准化。我们希望通过这一项目，以系统化、可伸缩的方式实施多年来在这个领域传授的方法，以便利用算法交易领域的许多机遇。近来，此类项目即便对于像我们这样规模较小的团队来说也是可能的，这应该归功于 Python 以及数据驱动和人工智能优先金融学。

我以如下的话作为第 1 版前言的结束：

> 对于 Python 确立金融行业中重要技术地位这一事实，我确实感到很兴奋。我敢肯定，它在未来将会起到更重要的作用，例如在衍生品和风险分析或者高

> 性能计算领域中。我希望本书能够帮助专业人士、研究人员和学生在面对这一迷人领域中的挑战时，最大限度地利用 Python。

当我在 2014 年写下上述文字时，还无法预测 Python 在金融学中将会变得多么重要。2018 年，我因为自己的期许和希望已经被大大超越而感到快乐。也许，本书的第 1 版对此起到了小小的作用。无论如何，读者都应该感谢不懈努力的开源开发者们，如果没有他们，就不可能书写 Python 的成功故事。

本书约定



这个元素表示提示或者建议。



这个元素表示一般注解。



这个元素表示警告或者注意事项。

代码示例的使用

补充材料（特别是 Jupyter Notebook 和 Python 脚本/模块）可以从 Python Quants 下载。

本书的目的是帮助读者完成工作。一般而言，你可以在你的程序和文档中使用本书中的代码，而且也没有必要取得我们的许可。但是，如果你要复制的是核心代码，则需要和我们打个招呼。例如，你可以在无须获取我们许可的情况下，在程序中使用本书中的多个代码块。但是，销售或分发 O'Reilly 图书中的代码则需要取得我们的许可。通过引用本书中的示例代码来回答问题时，不需要事先获得我们的许可。但是，如果你的产品文档中融合了本书中的大量示例代码，则需要取得我们的许可。

在引用本书中的代码示例时，如果能列出本书英文原书的属性信息是最好不过（但不是必需的）。属性信息通常包括书名、作者、出版社和 ISBN。例如："*Python for Finance*, 2nd Edition, by Yves Hilpisch (O'Reilly). Copyright 2019 Yves Hilpisch, 978-1-492-02433-0."

在使用书中的代码时，如果不确定是否属于正常使用，或是否超出了我们的许可，请通过 permissions@oreilly.com 与我们联系。

O’Reilly Safari

Safari（原 Safari Books Online）是供企业、政府、教育工作者和个人使用的会员制培训与参考平台。

会员有权访问来自 250 个出版商（包括 O’Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology）、数以千计的图书、培训视频、学习路径、交互教程和推荐书目。

更多信息请访问 O’Reilly 官网。

联系方式

如果你想就本书发表评论或有任何疑问，敬请联系出版社：

美国：

O’Reilly Media. Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

我们还为本书建立了一个网页（官方网站），其中包含了勘误表、示例和其他额外的信息。

关于本书的技术性问题或建议，请发邮件到：bookquestions@ oreilly.com。

欢迎登录我们的网站，查看更多我们的图书、课程、会议和最新动态等信息。

致谢

我要感谢所有为本书的实现提供帮助的人，特别是 O’Reilly 的团队，他们在许多方面改进了我的手稿。感谢审稿人休・布朗（Hugh Brown）和杰克・万托布拉斯（Jake VanderPlas），本书从他们有价值的反馈和许多建议中获益匪浅。余下的任何错误责任当然在我。

迈克尔・施韦德（Michael Schwed）曾与我紧密协作十多年的时间，值得我特别感谢。多年以来，我已经从他的工作、支持和 Python 技巧中得到多方面的好处。

我还要感谢 Refinitiv 公司（前 Thomson Reuters）的詹森·拉姆晶达尼（Jason Ramchandani）和豪尔赫·桑托斯（Jorge Santos），他们不仅持续支持我的工作，还为开放源码社区做出了贡献。

和第 1 版一样，本版也从我数年来参加的数十次“Python 金融”讨论以及数百小时的“Python 金融”培训中得到很大的益处。在许多情况下，来自参与者的反馈有助于改进我的培训材料，这些材料往往最终成为本书中的某个章节。

本书第 1 版的写作花费了大约一年的时间。总体上，编写和升级第 2 版也花了大约一年，这比我的预期要长一些，主要的原因是为了这个话题我经常忙于旅行和商务演讲。

写作需要许多独处的时间，无法与家人共度。因此，我要感谢桑德拉、莉丽、亨利、阿道夫、佩特拉和海因茨的理解和支持——不仅仅是在编写本书的时候。

和第 1 版一样，我将本书的第 2 版献给我可爱、坚强且富于同情心的妻子桑德拉。多年以来，她赋予家庭新的意义。感谢你。

Yves

德国萨尔州，2018 年 11 月

资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

配套资源

本书提供如下资源：

- 本书源代码；
- 书中彩图文件。

要获得以上配套资源，请在异步社区本书页面中点击配套资源，跳转到下载界面，按提示进行操作即可。注意：为保证购书读者的权益，该操作会给出相关提示，要求输入提取码进行验证。

如果您是教师，希望获得教学配套资源，请在社区本书页面中直接联系本书的责任编辑。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，点击“提交勘误”，输入勘误信息，点击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的 100 积分。积分可用于在异步社区兑换优惠券、样书或奖品。



扫码关注本书

扫描下方二维码，您将会在异步社区微信服务号中看到本书信息及相关的服务提示。

与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问 www.epubit.com/selfpublish/submission 即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于 2015 年 8 月，提供大量精品 IT 技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网 https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品 IT 专业图书的品牌，依托于人民邮电出版社近 30 年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的 LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

异步社区　　　　微信服务号

目录

第 1 部分　Python 与金融

第 2 部分　掌握基础知识

第 3 部分 金融数据科学

第 1 部分

Python 与金融

这一部分介绍 Python 的金融应用，包括两章。

- 第 1 章简短地讨论 Python 的总体情况，证明 Python 确实适用于处理金融行业和财务（数据）分析中遇到的技术难题。
- 第 2 章介绍 Python 基础架构和工具，目的是简洁地概述 Python 环境管理的重要知识，帮助你开始 Python 交互式金融分析及金融应用开发。

第 1 章

为什么将 Python 用于金融

银行本质上是技术公司。

——胡戈 · 班齐格

1.1 Python 编程语言

Python 是一种高级的多用途编程语言，广泛用于各种非技术和技术领域。在 Python 网站上，你可以找到如下行动纲领：

> Python 是一种具备动态语义、面向对象的解释型高级编程语言。它的高级内建数据结构与动态类型及动态绑定相结合，使其在快速应用开发上极具吸引力，也适合于作为脚本或者“黏合剂”语言，将现有组件连接起来。Python 简单、易学的语法强调可读性，因此可以降低程序维护成本。Python 支持模块和软件包，鼓励模块化和代码重用。Python 解释程序和大量标准库可以以源代码或者二进制的形式免费取得，它们用于所有主要平台，并且可以随意分发。

上述纲领很好地描述了 Python 成为当今主要编程语言之一的原因。当前，在学校、Web 公司、大型企业和金融机构以及任何科学领域，都有初学者和熟练的专业开发人员在使用 Python。

Python 有如下特征。

开放源码

> Python 和大部分可用的支持库及工具都是开源的，通常都有相当灵活和开放的许可证。

解释型

CPython 参考实现是该语言的一个解释程序，在运行时它将 Python 代码翻译为可执行字节代码。

多重范型

Python 支持不同的编程和实现范型，例如面向对象和命令式、函数式或者过程式编程。

多用途

Python 可以用于快速、交互式代码开发，也可以用于构建大型应用程序；它可以用于低级系统操作，也可以承担高级分析任务。

跨平台

Python 可用于大部分重要的操作系统，如 Windows、Linux 和 Mac OS；它用于构建桌面应用和 Web 应用；可以在庞大的群集和强大的服务器上使用，也可以在树莓派这样的小设备上运行。

动态类型

Python 中的类型通常在运行时推知，而不像大部分编译语言那样静态声明。

缩进感知

和大部分其他编程语言不同，Python 使用缩进标记代码块，缩进标记代替了圆括号、方括号或者分号。

垃圾收集

Python 具有自动垃圾收集机制，避免程序员来管理内存。

关于 Python 语法及其意义，Python 增强提案 20——即所谓的“Python 之禅”——提供了重要的指导方针。每个交互 shell 都可以用命令 `import this` 访问它：

```
In [1]: import this
        The Zen of Python, by Tim Peters

        Beautiful is better than ugly.
        Explicit is better than implicit.
        Simple is better than complex.
        Complex is better than complicated.
        Flat is better than nested.
        Sparse is better than dense.
        Readability counts.
        Special cases aren't special enough to break the rules.
        Although practicality beats purity.
```

```
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.1.1　Python 简史

Python 对于某些人来说可能还是个新事物，但是它已经出现了很长时间。实际上，早在 20 世纪 80 年代，荷兰人吉多・范・罗苏姆（Guido van Rossum）就开始了开发工作。他现在仍然活跃在 Python 开发中，被 Python 社区授予“仁慈独裁者”的称号。2018 年 7 月，罗苏姆在数十年积极推动 Python 核心开发工作之后让位。下面是 Python 开发的里程碑：

- 1991 年发行的 Python 0.9.0（第一个发行版本）;
- 1994 年发行的 Python 1.0；
- 2000 年发行的 Python 2.0；
- 2008 年发行的 Python 2.6；
- 2008 年发行的 Python 3.0；
- 2009 年发行的 Python 3.1；
- 2010 年发行的 Python 2.7；
- 2011 年发行的 Python 3.2；
- 2012 年发行的 Python 3.3；
- 2014 年发行的 Python 3.4；
- 2015 年发行的 Python 3.5；
- 2016 年发行的 Python 3.6；
- 2018 年 6 月发行的 Python 3.7。

值得注意的是，在开发之中有两个可用的主要版本，更重要的是，它们从 2008 年起并行使用，这有时候令 Python 初学者感到困惑。到本书编著之时，这种情况可能还将持续一段时间，因为仍有大量可用和生产代码是用 Python 2.6/2.7 编写的。本书第 1 版基于 2.7.x 版本，但第 2 版将全部使用 Python 3.7。

1.1.2 Python 生态系统

Python 作为一个生态系统，而不仅是一门编程语言，其主要特征是有大量可用的库和工具。这些库和工具通常必须在需要（例如，绘图库）时导入或者作为单独的系统进程（例如，Python 交互开发环境）启动。导入意味着使某个库可用于当前命名空间和当前 Python 解释程序进程。

Python 本身自带了一组大型的程序库和模块，它们叫作 Python 标准库，该标准库在不同方面增强了基本解释程序。例如，基本数学计算可以在不做任何导入的情况下完成，而更专业的数学函数必须通过 math 模块导入：

```
In [2]: 100 * 2.5 + 50
Out[2]: 300.0

In [3]: log(1)❶

        ---------------------------------------------------------------
        NameError                          Traceback (most recent call last)
        <ipython-input-3-74f22a2fd43b> in <module>
        ----> 1 log(1)❶

        NameError: name 'log' is not defined

In [4]: import math ❷

In [5]: math.log(1)❷
Out[5]: 0.0
```

❶ 没有进一步导入，就会发生错误。

❷ 导入 math 模块后，可以执行计算。

在任何安装中 math 都是可用的标准 Python 库，但是还有许多库是可选安装的，它们可以通过和标准模块相同的方式使用。这些库来自不同的（Web）来源。然而，通常建议使用某种 Python 程序包管理器，确保所有库相互一致（这个主题的更多内容参见第 2 章）。

目前为止介绍的代码示例都使用 Python 交互式环境：IPython 和 Jupyter。它们可能是本书写作时最广泛使用的 Python 交互式开发环境之一。虽然 IPython 刚开始出现时只是一个增强的 shell，但是现在它已经有了集成开发环境（IDE）的许多典型特性（例如，支持性能分析和调试）。IPython 中缺乏 Vim（也可与 IPython 集成）等高级文本/代码编辑器所提供的功能，因此，将 IPython 与某种文本/代码编辑器组合使用、组成 Python 开发过程基本工具集的情况很常见。

IPyhon 从许多方面增强了标准交互式 shell。它提供了改进的命令行历史功能，并且能够进行简单的对象检查。例如，在函数名称后添加一个？就可以打印函数的帮助文本（添加？？将提供更多信息）。

IPython 最初有两个流行版本：一个 shell 版本和一个基于浏览器的版本（Notebook）。Notebook 变体非常实用和流行，从而演变为一个独立的语言无关的项目 Jupyter。考虑到它的背景，Jupyter Notebook 继承了大部分 IPython 的有益特性也就不足为奇了，它还提供了更多的新功能，例如可视化。

使用 IPython 的更多细节请参见 VanderPlas（2016 版，第 1 章）。

1.1.3　Python 用户谱系

Python 不仅对专业软件开发人员有很大的吸引力，临时开发人员、领域专家和科研开发人员也使用它。

专业软件开发人员寻求高效构建大型应用程序所需的一切工具。Python 支持几乎所有编程范型，有强大的开发工具。从理论上说，Python 可以应对任何工作。这些类型的用户通常构建自己的框架和类，也依靠基础的 Python 和科学栈进行工作，并且尽最大的努力利用生态系统。

科研开发人员和领域专家通常频繁使用某些库和框架，构建他们长年改进和优化的应用程序，并且根据特定的需求调整生态系统。这组用户通常参与较长的交互式会话，快速建立新代码原型，并探索和可视化其研究及领域数据集。

临时开发人员喜欢在确定 Python 具有优势的特定问题上使用 Python。例如，访问 Matplotlib 的展示页面，复制 Matplotlib 提供的某一段可视化代码，根据特殊需求调整这些代码等。

Python 用户还有另一个重要的群体：编程入门者，也就是刚刚开始编程的人。现在，Python 在大学、专业院校甚至中小学校中已经成为向学生介绍编程的流行语言[1]。这种现象的主要原因之一是其基本语法即使对于非开发人员也很容易学习和理解。此外，Python 支持几乎所有编程风格[2]。

1.1.4　科学栈

某些库的集合被统称为科学栈（Scientific Stack），其中包括以下程序库。

NumPy

　NumPy 提供多维数组对象，以存储同构或者异构数据；它还提供操作这一数组对象的优化函数/方法。

SciPy

　SciPy 是子库和函数的集合，它可以实现科学或者金融中常常用到的重要标准功能；例如，你可以找到 3 次样条插值和数值积分的函数。

1　例如，Python 已经成为纽约城市大学博鲁克学院金融工程研究生课程中使用的主要语言之一。——原注

2　维基百科上可以找到许多开发人员和非开发人员学习 Python 入门知识的宝贵资源。——原注

Matplotlib

Matplotlib 是非常流行的 Python 绘图和可视化库，提供 2D 和 3D 可视化功能。

pandas

pandas 在 NumPy 基础上构建，可提供更丰富的时间序列和表格数据的管理与分析；它与 Matplotlib 在绘图上、与 PyTables 在数据存储和读取上紧密集成。

scikit-learn

scikit-learn 是流行的机器学习（ML）程序库，为许多不同的 ML 算法（如估值、分类或者聚类）提供统一的应用编程接口（API）。

PyTables

PyTables 是流行的 HDF5 数据存储库封装器；这个库可实现基于层次数据库/文件格式的优化磁盘 I/O 操作。

根据特定的领域或问题，科学栈可以通过更多的库进行扩展，这些库多半在一个或者多个上述的基本库基础上构建。但是，最小公分母（或称基本组成部分）通常是 NumPy ndarray 类（见第 4 章）和 pandas DataFrame 类（见第 5 章）。

仅从编程语言来讲，有许多其他语言在语法和简洁性上可与 Python 比肩。例如，Ruby 也是相当流行的语言，可与 Python 相提并论。在该语言的网站上，你可以找到如下的描述：

> 一种动态的开放源码编程语言，重视简洁性和效率。它具备简洁的语法，阅读自然、易于编写。

大部分 Python 使用者可能也赞成用相同的陈述描述 Python 本身。但是，对于许多 Python 用户而言，它与 Ruby 等同样具有吸引力的语言之间的区别在于科学栈。这使 Python 不仅是优秀、简洁的语言，还可以代替 Matlab 或者 R 等领域专用语言和工具集。此外，它默认提供各种人员（如熟练的 Web 开发人员或者系统管理员）所需要的任何功能。Python 还擅长与 R 等领域特定语言交互，因此，你所要做的决策通常不是“用 Python 还是其他语言？”，而是以哪一种语言为主。

1.2　金融中的科技

现在，我们对 Python 已经有了大致的认识，接下来简短地介绍一下科技在金融中的作用就很有意义了。这将使我们更好地评判 Python 在金融行业中已经承担的任务，更重要的是，还可以评判未来承担的任务。

在某种意义上，科技对于金融机构（例如与生物技术公司相比）或者财务部门（与其他企业职能部门相比，如后勤）没有什么特别的作用。然而，近年来，在创新和监管的刺激下，银行和其他金融机构（如对冲基金）越来越多地发展成为技术公司而不仅

仅是金融中介机构。科技成为了全球几乎所有金融机构的重要资产，具备导致竞争优势和劣势的潜力。某些背景信息可以解释这种发展的原因。

1.2.1 科技投入

银行和金融机构共同组成了每年在科技上投入最多的行业。因此，下面的陈述不仅说明科技对金融行业的重要性，也说明了金融行业对科技的重要性：

> 美国马萨诸塞州弗雷明汉，2018 年 6 月 14 日——根据国际数据公司（IDC）的一系列“金融服务 IT 开销指南”称，全球金融服务公司在信息科技（IT）上的开销将从 2018 年的 4400 亿美元增长到 2021 年的近 5000 亿美元。
>
> ——IDC

特别是，银行和其他金融机构正在参与业务及运营模式数字化的竞争：

> 2017 年，北美地区银行在新科技上的开销高达 199 亿美元。
>
> 银行开发当前系统并致力于新的技术解决方案，以增强其在全球市场上的竞争力，吸引对新的网络和移动科技感兴趣的客户。对于为银行业提供新思路和软件解决方案的全球金融科技公司来说，这是一个巨大的机遇。
>
> ——Statista

当今的大型跨国银行通常雇佣数千名开发人员，以维护现有系统并构建新系统。具有大量科技需求的大型投资银行每年的科技预算往往达到数十亿美元。

1.2.2 作为业务引擎的科技

科技发展对金融行业的创新和效率增进也有贡献。通常，这一领域的项目统称为“数字化项目”：

> 金融服务业在过去数年中已经历了科技先导的彻底变革。许多高管期望 IT 部门改进效率，促进游戏规则的创新——同时降低成本，继续支持遗留系统。与此同时，金融科技初创企业正在逐步蚕食原有市场，提出对客户友好的解决方案，这些方案是从头开始设计的，不受遗留系统的阻碍。
>
> ——普华永道第 19 次年度全球 CEO 调查报告（2016 年）

效率提高的副作用之一就是，金融机构往往必须在更为复杂的产品或者交易中寻求竞争优势。这当然会使风险增大，并使风险管理、监控和监管越来越困难。2007 年和 2008 年的金融危机说明了这些发展带来的潜在危险。同样，“算法和计算机失控”也给金融市场带来潜在的风险。2010 年 5 月的所谓“闪电崩盘”事件戏剧性地展现了上述风险，自动化卖出导致某些股票和股票指数在当日大幅度下跌。本书第 4 部分将介绍与金融工具算法交易相关的内容。

1.2.3　作为进入门槛的科技和人才

一方面，在其他条件不变的情况下，随着时间的推移，科技的进步会降低成本。另一方面，金融机构持续在科技上投入巨资，以增大市场份额、保持自身地位。在今天的金融市场上取得一席之地往往需要在科技和熟练人员上大规模投资。考虑衍生品分析领域的一个例子：

> 在整个软件生命期中，采用内部 OTC（衍生品）定价策略的公司仅在一个完整的衍生品库的构建、维护和改进上就需要投入 2500 万 ~ 3600 万美元。
>
> ——Ding 2010

构建一个完善的衍生品分析库不仅成本高昂和费时，而且需要有足够的专业人士来进行这项工作。这些专业人士必须有正确的工具和技术，才能完成相应的任务。随着 Python 生态系统的发展，与 10 年前相比，这些工作变得更加高效，相关的成本也有了显著的下降。第 5 部分介绍衍生品分析，仅用 Python 及标准库就构建了一个规模虽小却强大且灵活的衍生品定价库。

下面，我们引用另一段美国长期资本管理公司（LTCM）的陈述，以进一步支持关于科技和人才的观点。LTCM 曾是最受尊敬的计量对冲基金，但是在 20 世纪 90 年代末破产：

> Meriwether 在最新型计算机系统上花费了 2000 万美元，并雇用一个由一流金融工程师成的团队在 LTCM 操纵该系统，他们在康涅狄格州的格林尼治开始工作。这是行业级别的风险管理。
>
> ——Patterson 2010

Meriwether 花费数百万美元才能得到的计算能力，在今天只需要几千美元就能实现。第 2 章介绍了在云中建立交互式金融分析、应用开发和 Python 部署的方法。这种专业基础设施每月的起始费用仅为几美元。另一方面，大型金融机构的交易、定价和风险管理已经变得非常复杂，导致现在必须部署具有数万个计算核心的 IT 基础架构。

1.2.4　不断提高的速度、频率和数据量

金融行业有一个方面非常受科技进步的影响：金融交易决策和执行的速度及频率。Lewis 最近的著作（2014）生动而详细地描述了所谓的“闪电交易”——也就是以可能的最高速度进行的交易。

一方面，可用数据的时间标度越来越小，使实时反应成为必需的能力。另一方面，交易的速度和频率的提高使数据量进一步增大。这两方面相互补充，推动了金融交易平均时间标度的系统性下降。这种趋势在 10 年前就已经开始：

> 复兴资本公司的“大奖章”基金在 2008 年获得了 80%的惊人增长率，它以闪电般速度的计算机抓住了市场极端活跃的机会。Jim Simons 是当年世界盈利最高的对冲基金管理人，收入达到 25 亿美元。
>
> ——Patterson 2010

单只股票 30 年的每日股价数据大致包含 7500 个报价。这类数据是大部分现代金融理论的基础。例如，现代投资组合理论（MPT）、资本定价模型（CAPM）和风险价值（VaR）等理论都以每日股价数据为基础。

相比之下，苹果公司（AAPL）典型交易日报价次数大约为 15000——两倍于 30 年日终报价的数量（参见 1.4 节的例子）。这带来了许多挑战。

数据处理

只考虑和处理股票或者其他金融工具的日终报价是不够的。在每周的 7 天、每天的 24 小时内，某些金融工具上发生的事情“太多了”。

分析速度

决策往往必须在几毫秒甚至更短的时间内做出，有必要构建独自的分析能力工具，并实时分析大量数据。

理论基础

虽然传统金融理论和概念远称不上完美，但是它们经受了时间的考验（有些时候受到排斥）；对于毫秒级计量能力很重要的今天，仍然缺乏在很长时间内证明是稳定的一致性概念和理论。

总体上，上述挑战都只能由现代科技应对。令人有些惊讶的是，缺乏一致性理论的问题也常常通过技术方法处理。在这种情况下，高速算法利用的是市场微观结构要素（例如，订单流、买卖价差），而不依赖于某种金融推理方法。

1.2.5 实时分析的兴起

金融行业中有一个学科的重要性正在强劲增长：金融和数据分析。这种现象与行业中速度、频率和数据率飞速增长有紧密的关系。实际上，实时分析可以视为行业对这种趋势的反应。

粗略地讲，“金融和数据分析”指的是将应用软件、科技，与（可能是先进的）算法、数据收集、数据处理及分析方法相结合，以获得深刻的洞察力、做出决策或者满足监管需求的学科。这类分析的例子包括银行零售部门中某个金融产品定价结构的变化对销售情况影响的估算。另一个例子是投资银行衍生品复杂投资组合信用价值调整（CVA）的大规模隔夜计算。

金融机构在这种环境下主要面对两种挑战。

大数据

在“大数据”这一术语出现之前，银行和其他金融机构就必须处理海量数据。然而，单一分析任务所处理的数据量随着时间的推移而有了很大的增长，从而要求计算能力有所提高并有更大的内存与存储能力。

> 过去，决策者只能依赖结构化的定期计划、决策和风险管理过程，而今天决策者面对的是实时完成这些任务的需求。过去在后台通过隔夜批量运行可以完成的任务，现在已经转向前台实时执行。

同样，人们可以观察到科技和金融/商业发展之间的相互作用。一方面，市场上出现了通过现代科技的应用不断改进分析方法的速度和效率的需求。另一方面，科技的进步使几年甚至几个月之前认为不可能（或者由于预算约束而不可行）的新分析方法成为可能。

分析领域的一个重要趋势是利用中央处理单元（CPU）的并行结构和通用计算图形处理单元（GPGPU）的大规模并行结构。现在的 GPGPU 往往有 1000 多个计算核心，有时候有必要彻底反思并行性对不同算法的意义。用户通常必须学习新的范型和技术才能利用这种硬件，这仍是障碍。

1.3　用于金融的 Python

1.2 节介绍了科技在金融中发挥作用的一些领域：

- 金融行业中的科技成本；
- 作为新业务和创新业务引擎的科技；
- 作为金融行业进入门槛的科技；
- 不断提升的速度、频率和数据量；
- 实时分析的兴起。

本节，我们分析 Python 如何帮助你应对这些方面的多种挑战。不过首先让我从更为基础的方面——语言和语法介绍用于金融的 Python。

1.3.1　金融和 Python 语法

在金融环境中迈出使用 Python 第一步的大部分人都可能要攻克某个算法问题。这和想要解出微分方程、求取积分或者可视化某些数据的科学工作者类似。一般来说，在这一阶段，对正规开发过程、测试、文档或者部署没有太多的要求。然而，这一阶段似乎是人们特别容易爱上 Python 的时候，主要原因是 Python 的语法总体上和用于描述科学问题或者金融算法的数学语法相当接近。

我们可以通过一个简单的金融算法——通过蒙特卡洛模拟方法估计欧式看涨期权的价值来说明这一现象。我们将考虑 Black-Scholes-Merton（BSM）模型，在这种模型中期权的潜在风险遵循几何布朗运动。

假定我们使用以下数值化参数进行估值：

- 初始股票指数水平 S_0=100；
- 欧式看涨期权的行权价格 K=105；
- 到期时间 T=1 年；
- 固定无风险短期利率 r=5%；
- 固定波动率 σ=20%。

在 BSM 模型中，到期指数水平是一个随机变量，由公式 1-1 给出，其中 z 是一个标准正态分布随机变量。

公式 1-1　Black-Scholes-Merton（1973）到期指数水平

$$S_T = S_0 \exp\left(\left(r - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}z\right)$$

下面是蒙特卡洛估值过程的算法描述。

（1）从标准正态分布中取得 I 个（伪）随机数 $z(i)$，$i\in\{1, 2, \cdots, I\}$。

（2）为给定的 $z(i)$和公式 1-1 计算所有到期指数水平 $S_T(i)$。

（3）计算到期时期权的所有内在价值 $h_T(i)=\max(S_T(i)-K,0)$。

（4）通过公式 1-2 中给出的蒙特卡罗估算函数估计期权现值。

公式 1-2　欧式期权的蒙特卡洛估算函数

$$C_0 \approx \mathrm{e}^{-rT}\frac{1}{I}\sum_I h_T(i)$$

现在，我们需要将这个问题和算法翻译为 Python 代码。下面的代码将实现一些必要的步骤。

```
In [6]: import math
        import numpy as np ❶

In [7]: S0 = 100. ❷
        K = 105. ❷
        T = 1.0 ❷
        r = 0.05 ❷
        sigma = 0.2 ❷

In [8]: I = 100000 ❷

In [9]: np.random.seed(1000) ❸
```

```
In [10]: z = np.random.standard_normal(I) ❹

In [11]: ST = S0 * np.exp((r - sigma ** 2 / 2) * T + sigma * math.sqrt(T) * z) ❺

In [12]: hT = np.maximum(ST - K, 0) ❻

In [13]: C0 = math.exp(-r * T) * np.mean(hT) ❼

In [14]: print('Value of the European call option: {:5.3f}.'.format(C0)) ❽
         Value of the European call option: 8.019.
```

❶ NumPy 在这里作为主程序包使用。

❷ 定义模型并模拟参数值。

❸ 随机数生成器种子值固定。

❹ 提取标准正态分布随机数。

❺ 模拟期末价值。

❻ 计算期权到期收益。

❼ 计算蒙特卡洛估算函数。

❽ 打印输出估算结果。

以下 3 个方面值得注意。

语法

　　Python 语法与数学语法相当接近，例如参数赋值的方面。

翻译

　　每条数学或者算法语句一般都可以翻译为单行 Python 代码。

向量化

　　NumPy 的强项之一是紧凑的向量化语法，例如，允许在单一代码行中进行 10 万次计算。

这段代码可以用于 IPython 或 Jupyter Notebook 等交互式环境。但是，需要频繁重用的代码一般组织为所谓的模块（或者脚本），也就是带有.py 后缀的 Python（文本）文件。本例的模块如例 1-1 所示，可以将其保存为名为 bsm_msc_euro.py 的文件。

例 1–1　欧式看涨期权的蒙特卡洛估值

```
#
# Monte Carlo valuation of European call option
```

```
# in Black-Scholes-Merton model
# bsm_mcs_euro.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import math
import numpy as np

# Parameter Values
S0 = 100. # initial index level
K = 105. # strike price
T = 1.0 # time-to-maturity
r = 0.05 # riskless short rate
sigma = 0.2 # volatility

I = 100000 # number of simulations

# Valuation Algorithm
z = np.random.standard_normal(I) # pseudo-random numbers
# index values at maturity
ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * math.sqrt(T) * z)
hT = np.maximum(ST - K, 0) # payoff at maturity
C0 = math.exp(-r * T) * np.mean(hT) # Monte Carlo estimator

# Result Output
print('Value of the European call option %5.3f.' % C0)
```

这一小节中的简单算法示例说明，Python 的基本语法很适合为经典的科学语言二重奏——英语和数学来提供补充。在科学语言组合中添加 Python 能使其更加全面。我们现在拥有：

- 用于写作和谈论科学、金融等问题的英语；
- 用于简洁、精确地描述抽象特征、算法、复数等并为其建模的数学；
- 从技术上建立抽象特征、算法、复数等的模型并加以实现的 Python。

数学和 Python 语法

几乎没有任何编程语言像 Python 这样接近数学语法。因此，数值算法很容易从数学表示翻译为 Python 实现。通过 Python，我们可以在这些领域中高效地进行原型化、开发和代码维护。

在某些领域中，使用伪代码是常见的做法，这引入了第 4 个语言家族成员。举个例子，

伪代码的任务是以更技术性的方式表示金融算法，不但与数学表示接近，而且还接近于技术实现。除了算法本身，伪代码还考虑了计算机的工作原理。

采用这种方法一般是因为，使用大部分编程语言时，技术实现和正式的数学表现形式的距离相当“遥远”。大部分编程语言都必须包含许多只在技术上需要的元素，但在数学和代码中很难看到等价的元素。

时下，Python 常常以伪代码的方式被使用，因为它的语法和数学很类似，而且技术“开销”可以控制到最低。这一点是通过该语言所体现的一些高层概念实现的，这些概念不仅有其优势，也带来了风险和其他代价。不过可以肯定，我们可在需求出现的时候使用 Python，从一开始就遵循其他语言可能需要的严格实现和编码方法。从这个意义上说，Python 可以在两个世界：高层次的抽象和严格的实现中提供最佳的平衡。

1.3.2　Python 的效率和生产率

从较高的层次看，使用 Python 的好处可以从以下 3 个维度衡量。

效率

　　Python 如何更快地获得结果、节约成本、节约时间？

生产率

　　Python 如何在相同的资源（人员、资产等）下完成更多的工作？

质量

　　Python 能够让我们做哪些替代技术所不能做到的事情？

对这些特性的讨论当然不可能很全面。然而，可以将某些特性作为出发点。

1. 在更短的时间里得到成果

Python 效率较为明显的领域之一是交互式的数据分析。这些领域从 IPython、Jupyter Notebook 等有力工具和 pandas 之类的程序中库获益良多。

假设你是一位正在撰写硕士论文的金融专业学生，对标普 500 指数感兴趣，想要分析 1 年的历史指数水平，以了解指数在这段时间内的波动性，你希望找到证据证明这种变动性与某些典型的模型假设相反，它是随时间变动而非固定。而且，应该对结果进行可视化，你要进行的主要的工作如下：

- 从网络上下载指数水平数据；
- 计算年化对数收益率的滚动标准差（波动率）；
- 绘制指数水平数据和波动性结果图表。

这些任务很复杂，在不久之前还被认为是专业金融分析师才能完成的。而在今天，即

使是金融专业的学生也可以轻松地对付这类问题。我们来看看具体的做法——此时还不用操心语法的细节（后续的章节将对所有细节进行解释）：

```
In [16]: import numpy as np ❶
         import pandas as pd ❶
         from pylab import plt, mpl ❷

In [17]: plt.style.use('seaborn') ❷
         mpl.rcParams['font.family'] = 'serif' ❷
         %matplotlib inline

In [18]: data = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                           index_col=0, parse_dates=True) ❸
         data = pd.DataFrame(data['.SPX']) ❹
         data.dropna(inplace=True) ❹
         data.info() ❺
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2138 entries, 2010-01-04 to 2018-06-29
         Data columns (total 1 columns):
         .SPX    2138 non-null float64
         dtypes: float64(1)
         memory usage: 33.4 KB

In [19]: data['rets'] = np.log(data / data.shift(1)) ❻
         data['vola'] = data['rets'].rolling(252).std() * np.sqrt(252) ❼
         In [20]: data[['.SPX', 'vola']].plot(subplots=True, figsize=
        (10, 6)); ❽
```

❶ 导入 NumPy 和 pandas。

❷ 导入 matplotlib 并配置 Jupyter 绘图样式和方法。

❸ pd.read_csv()可以读取远程或者本地存储的逗号分隔值（CSV）形式数据集。

❹ 选取一个数据子集，删除 NaN（非数值）值。

❺ 显示关于数据集的一些元信息。

❻ 以向量化的方式（在 Python 级别上“无循环”）计算对数收益率。

❼ 得出滚动年化波动率。

❽ 最后绘制两个时间序列。

图 1-1 展示了这一简短交互会话所得到的图形化结果。用几行代码就足以完成金融分析中的典型复杂任务：数据收集、复杂和重复的数学计算以及结果的可视化，令人觉得不可思议。从这个例子中可以看到，pandas 使整个时间系列的处理变得就像浮点数上的数学运算那样容易。

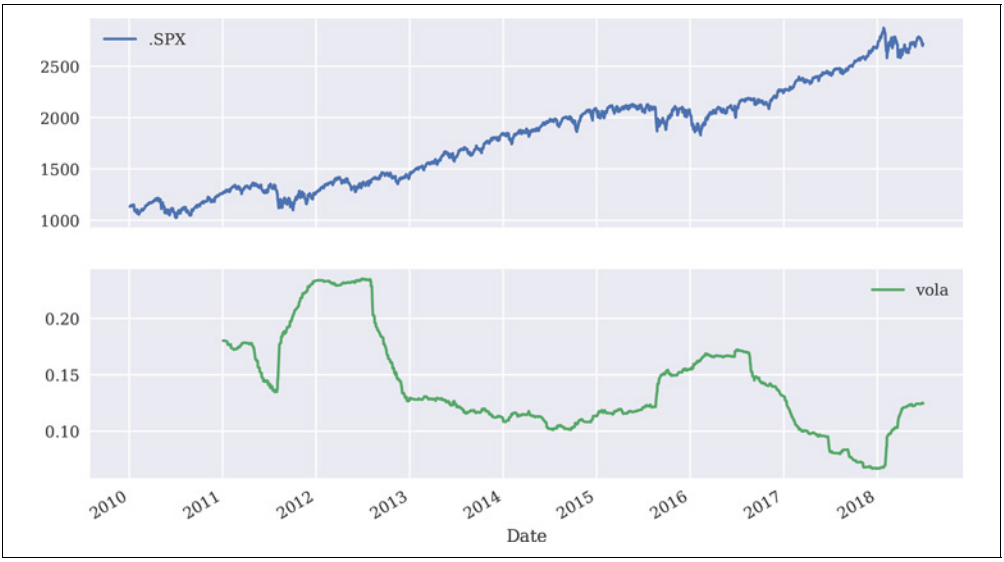


图 1-1　标普 500 收盘价和年化波动率

将这个例子转换到专业的金融环境中，可以看出金融分析师在应用提供高层次抽象的合适的 Python 工具和库的时候，能够将焦点放在自身的领域上，而不用关心复杂的技术细节。分析师可以快速反应，几乎实时地提供宝贵的洞见，确保自己比竞争对手先行一步。这种效率的提高很容易转换为可度量的财务效果。

2. 确保高性能

一般来说，Python 的语法相当简洁，编码效率相对高是为人们所接受的说法。但是，由于 Python 本质上是解释型语言，因此存在一种偏见，认为 Python 对于金融学中的计算密集任务来说速度过于缓慢。确实，在某些特定的实现方法下，Python 可能确实很慢，但是，它并不一定都那么缓慢——它可以在几乎所有应用领域中表现出高性能。理论上，人们至少可以找到 3 种提高性能的策略。

惯例和范型

　　一般来说，Python 可以用许多不同的方式得出相同的结果，但是这些方式的性能特性有相当大的区别；只要选择合适的方式（如特定的实现方法、明智地使用数据结构、通过向量化避免循环或者使用 pandas 等特定库），就可以显著地改善效果。

编译

　　现在，有许多高性能库可以提供重要函数的编译版本，或者将 Python 代码静态或者动态地（在运行时或者调用时）编译为机器代码，这种代码的速度比纯 Python 代码要快好几个数量级。比较流行的高性能库有 Cython 和 Numba 等。

并行化

许多计算任务（特别是金融学中的计算任务）可以从并行执行中得到很大好处；这对 Python 来说不足为奇，可以轻松地实现。

使用 Python 实现高性能计算

Python 本身不是一种高性能计算技术。但是，Python 已经发展成为一种访问当前高性能技术的理想平台。在这个意义上，Python 已经成为高性能计算的“黏合剂”语言。

本节坚持使用简洁实用的例子介绍上述 3 种策略（后续的章节将详细介绍这些策略）。金融分析中相当常见的任务之一是在大量数字上计算复杂的数学表达式。在这方面，Python 本身就提供了所有必需的功能：

```
In [21]: import math
         loops = 2500000
         a = range(1, loops)
         def f(x):
             return 3 * math.log(x) + math.cos(x) ** 2
         %timeit r = [f(x) for x in a]
         1.59 s ± 41.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Python 解释程序在本例中需要 1.6s 的时间来完成 250 万次函数 f 的计算。使用 NumPy 可以完成相同的任务，它提供了优化（也就是预先编译）的函数来处理这种基于数组的运算：

```
In [22]: import numpy as np
         a = np.arange(1, loops)
         %timeit r = 3 * np.log(a) + np.cos(a) ** 2
         87.9 ms ± 1.73 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

NumPy 可以将执行时间大幅缩减到约 88ms。甚至有一个库是专门用于此类任务的，这个库叫作 numexpr，得名于“数值表达式”（Numerical Expressions）。它可以编译表达式来改善 NumPy 通用功能的性能。例如，在执行期间避免 ndarray 对象在内存中复制：

```
In [23]: import numexpr as ne
         ne.set_num_threads(1)
         f = '3 * log(a) + cos(a) ** 2'
         %timeit r = ne.evaluate(f)
         50.6 ms ± 4.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

使用这种更特殊的方法可以进一步将执行时间降低至约 50ms。而且，numexpr 还内建并行执行单独运算的功能。这使我们能够使用一个 CPU 的多个线程：

```
In [24]: ne.set_num_threads(4)
         %timeit r = ne.evaluate(f)
```

```
22.8 ms ± 1.76 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

本例利用了 4 个线程，执行时间进一步下降到 23ms，总体的性能提升了 90 倍。特别要注意的是，这类改善不需要修改基本问题/算法，也不用了解任何有关编译和并行化问题的知识，即使是非专业人士也可以从较高的层次上去利用这种功能。当然，前提是你必须知道存在这些功能。

这个例子说明，Python 提供了一些选项可以更好地利用现有资源，也就是提高生产率。利用并行化方法，可以在同样的时间里完成 3 倍于串行化方法的计算，这只需要告诉 Python 使用多个 CPU 线程（而非仅使用一个线程）即可。

1.3.3 从原型化到生产

从执行速度的角度看，交互式分析的效率和性能当然是 Python 值得考虑的两个好处。在金融学中使用 Python 的另一个好处初看似乎比较不起眼，但是细看之下就会发现它本身是一个重要的战略因素。这就是以端到端（从原型化到生产）的方式使用 Python 的可能性。

当今全球金融机构的金融开发过程往往是一个分离的两步式过程。一方面，量化分析师（quant，也称宽客）负责模型开发和技术原型化。他们喜欢使用 MatLab 和 R 等工具和环境实现快速、交互式的应用程序开发。在开发的这一阶段，性能、稳定性、异常管理、数据访问分离和分析等问题都不重要。人们的主要目标是概念或者原型的验证，原型用于展现某种算法或者整个应用程序必需的主要功能。

一旦原型完成，IT 部门中的开发人员接管工作，他们负责将现有原型代码翻译为可靠、易于维护和高性能的生产代码。这一阶段中，通常在用于满足生产性能的 C++或者 Java 语言中有一个范型转换的过程。而且，正规的开发过程还要用到专业工具、版本控制等技术。

这种两步式方法会产生一些意外的结果。

效率低下

原型代码不能重用；算法必须实现两次；多余的工作消耗时间和资源；转译时产生的风险。

多种技能集

不同部门展现不同的技能集合，使用不同的语言实现“相同的工作”。

遗留代码

代码必须以不同的语言存在和维护，并使用不同的实现风格（例如，从架构的观点上看）。

另一方面，使用 Python 可以实现合理化的端到端过程——从最初的交互式原型化步骤到高可靠性、易于维护的生产代码。不同部门之间的沟通变得更加简单，工作人员的

培训也更为合理，只有一种主要语言覆盖金融应用构建的所有领域，还避免了在开发过程不同步骤中使用不同技术造成的低效和冗余性。总而言之，Python 可以为金融应用开发和算法实现提供一致性的技术框架。

1.4 数据驱动和人工智能优先的金融学

在 2018 年 8 月更新第 2 版的这一节时，2014 年本书第 1 版中有关科技与金融学之间关系的言论似乎仍然适用且重要。但是，本节将讲述金融业的两个重要趋势，它们将从根本上重塑这个行业。这两个趋势主要是在过去的几年内形成的。

1.4.1 数据驱动金融学

最重要的一些金融理论（如 MPT 和 CAPM）可以追溯到 20 世纪 50 年代和 60 年代。然而，它们仍然是经济、财务、金融工程和商业管理领域教育的基石。这可能令人惊讶，因为大部分此类理论的实证支持相当贫乏，而证据往往完全与理论相反。另一方面，它们的流行性也是可以理解的，因为它们接近于人们对金融市场表现的期望，而且，它们是建立在一些吸引人（通常过于简单）的假设之上的简洁数学理论的。

科学方法（如物理学）始于数据（如来自试验或者观察的数据），然后得出假设和理论，并用数据加以测试。如果测试结果是肯定的，那可以对这些假设和理论进行提炼，并以合适的方法记录下来，例如以研究型论文的方式发表。如果测试结果是否定的，假设和理论将被抛弃，并开始寻找复合数据的新理论。由于物理定律在一段时间里是稳定的，所以一旦发现并对齐进行了完备的测试，那么在最佳状况下，通常可以认为是永远成立的。

（计量）金融学的历史大部分与科学方法相悖。在许多情况下，理论和模型是在简化的数学假设下“从头”开发的，目标是发现金融学核心问题的简练回答。金融学的流行假设是，金融工具的回报呈标准正态分布，各种利率之间呈线性关系。由于这些现象在金融市场上很难看到，这些简单理论的实证往往很缺乏，也就不值得惊讶了。许多金融理论和模型都是首先提出公式、证明并发表，然后才进行实证测试的。在某种程度上，这当然是因为 20 世纪 50 年代或者 70 年代甚至更晚的时候，还不存在合适形式的金融数据，而今天就连攻读金融学士学位的学生都能得到这些数据。

20 世纪 90 年代初期到中期，金融机构所能得到的此类数据急剧增加。今天，甚至进行金融研究或者参与算法交易的个人都可以得到大量的历史数据，也可以通过流服务得到实时数据。这使我们可以专注于科学方法，科学方法通常从数据开始，然后得出思路、假设、模型和策略。

用一个简单的例子就可以说明，如今在本地机器上利用 Python 和 Eikon Data APIs 订阅大规模专业数据有多么简单。下面的例子是读取的苹果公司股票常规交易日中一个小时的分笔交易数据。读取的报价数据（包括交易量信息）大约有 15000 条。由于股票

代码为 AAPL，所以路透金融工具代码（RIC）为 AAPL.0：

```
In [26]: import eikon as ek ❶

In [27]: data = ek.get_timeseries('AAPL.O', fields='*',
                                  start_date='2018-10-18 16:00:00',
                                  end_date='2018-10-18 17:00:00',
                                  interval='tick') ❷

In [28]: data.info() ❷
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 35350 entries, 2018-10-18 16:00:00.002000 to
         2018-10-18 16:59:59.888000
         Data columns (total 2 columns):
         VALUE 35285 non-null float64
         VOLUME 35350 non-null float64
         dtypes: float64(2)
         memory usage: 828.5 KB

In [29]: data.tail() ❸
Out[29]: AAPL.O                VALUE VOLUME
         Date
         2018-10-18 16:59:59.433 217.13 10.0
         2018-10-18 16:59:59.433 217.13 12.0
         2018-10-18 16:59:59.439 217.13 231.0
         2018-10-18 16:59:59.754 217.14 100.0
         2018-10-18 16:59:59.888 217.13 100.0
```

❶ Eikon Data API 需要订阅和 API 连接才能使用。

❷ 读取苹果公司（AAPL.O）股票分笔数据。

❸ 显示最后 5 行数据。

Eikon Data API 不仅可以访问结构化金融数据（如历史报价数据），还可以访问新闻等非结构化数据。下面的例子读取一小部分新闻的元数据，并以全文本方式显示其中一篇文章开头的内容。

```
In [30]: news = ek.get_news_headlines('R:AAPL.O Language:LEN',
                                      date_from='2018-05-01',
                                      date_to='2018-06-29',
                                      count=7) ❶

In [31]: news ❶
Out[31]:
                                        versionCreated \
    2018-06-28 23:00:00.000 2018-06-28 23:00:00.000
```

```
    2018-06-28 21:23:26.526 2018-06-28 21:23:26.526
    2018-06-28 19:48:32.627 2018-06-28 19:48:32.627
    2018-06-28 17:33:10.306 2018-06-28 17:33:10.306
    2018-06-28 17:33:07.033 2018-06-28 17:33:07.033
    2018-06-28 17:31:44.960 2018-06-28 17:31:44.960
    2018-06-28 17:00:00.000 2018-06-28 17:00:00.000

                                                                 text \
    2018-06-28 23:00:00.000 RPT-FOCUS-AI ambulances and robot doctors: Chi...
    2018-06-28 21:23:26.526 Why Investors Should Love Apple's (AAPL) TV En...
    2018-06-28 19:48:32.627 Reuters Insider - Trump: We're reclaiming our ...
    2018-06-28 17:33:10.306 Apple v. Samsung ends not with a whimper but a...
    2018-06-28 17:33:07.033 Apple's trade-war discount extended for anothe...
    2018-06-28 17:31:44.960 Other Products: Apple's fast-growing island of...
    2018-06-28 17:00:00.000 Pokemon Go creator plans to sell the tech behi...
                                                                 storyId \
    2018-06-28 23:00:00.000 urn:newsml:reuters.com:20180628:nL4N1TU4F8:6
    2018-06-28 21:23:26.526 urn:newsml:reuters.com:20180628:nNRA6e2vft:1
    2018-06-28 19:48:32.627 urn:newsml:reuters.com:20180628:nRTV1vNw1p:1
    2018-06-28 17:33:10.306 urn:newsml:reuters.com:20180628:nNRA6e1oza:1
    2018-06-28 17:33:07.033 urn:newsml:reuters.com:20180628:nNRA6e1pmv:1
    2018-06-28 17:31:44.960 urn:newsml:reuters.com:20180628:nNRA6e1m3n:1
    2018-06-28 17:00:00.000 urn:newsml:reuters.com:20180628:nL1N1TU0PC:3

                            sourceCode
    2018-06-28 23:00:00.000   NS:RTRS
    2018-06-28 21:23:26.526 NS:ZACKSC
    2018-06-28 19:48:32.627   NS:CNBC
    2018-06-28 17:33:10.306 NS:WALLST
    2018-06-28 17:33:07.033 NS:WALLST
    2018-06-28 17:31:44.960 NS:WALLST
    2018-06-28 17:00:00.000   NS:RTRS
In [32]: story_html = ek.get_news_story(news.iloc[1, 2]) ❷

In [33]: from bs4 import BeautifulSoup ❸

In [34]: story = BeautifulSoup(story_html, 'html5lib').get_text() ❹

In [35]: print(story[83:958]) ❺
    Jun 28, 2018 For years, investors and Apple AAPL have been beholden to
    the iPhone, which is hardly a negative since its flagship product is
    largely responsible for turning Apple into one of the world's biggest
    companies. But Apple has slowly pushed into new growth areas, with
    streaming television its newest frontier. So let's take a look at what
    Apple has planned as it readies itself to compete against the likes of
    Netflix NFLX and Amazon AMZN in the battle for the new age of
```

```
entertainment.Apple's second-quarter revenues jumped by 16% to reach
$61.14 billion, with iPhone revenues up 14%. However, iPhone unit sales
climbed only 3% and iPhone revenues accounted for over 62% of total Q2
sales. Apple knows this is not a sustainable business model, because
rare is the consumer product that can remain in vogue for decades. This
is why Apple has made a big push into news,
```

❶ 读取一小部分新闻的元数据。

❷ 读取一篇文章的全部文本，形式为 HTML 文档。

❸ 导入 BeautifulSoup HTML 解析库并进行后续操作。

❹ 以纯文本方式（str 对象）提取内容。

❺ 打印新闻的开头。

尽管这两个例子很浅显，但它们说明，通过 Python 包装器库和数据订阅服务，可以以标准化、高效的方式得到结构化和非结构化的金融历史数据。在许多情况下，个人甚至可以使用 FXCM 集团、LLC 等交易平台（在第 14 章和第 16 章中会讲到）来免费获得类似的数据集。一旦在 Python 级别上得到数据（独立于原始来源），就可以利用 Python 数据分析生态系统的全部功能了。

> **数据驱动金融学**
>
> 近年来，数据成了金融业的推动力。即使最大、最成功的对冲基金也自称是“数据驱动”的，而非“金融驱动”的。越来越多的服务产品向大小机构及个人提供海量数据。Python 通常被选为与这些 API 交互、处理和分析数据的编程语言。

1.4.2　人工智能优先金融学

由于通过编程 API 能够取得大量金融数据，所以在金融问题（如算法交易）上应用人工智能（AI）方法、特别是机器与深度学习（ML，DL）就变得非常容易，也更有成果了。

Python 可以称得上 AI 界的“宠儿”。它往往被 AI 研究人员和从业者选为编程语言。从这个意义上说，金融领域从不同领域的发展中得益，这些领域有时甚至与金融没有太大的联系。举个例子，深度学习所用的 TensorFlow 开源程序库是 Google 公司开发和维护的，用于其母公司 Alphabet 自动驾驶汽车的制造和销售上。

TensorFlow 当然和股票自动算法交易一点关联都没有，但却可以用于预测金融市场的变动。第 15 章提供了许多这方面的例子。最广泛使用的 Python ML 程序库是 scikit-learn。下面的代码以高度简化的方式说明：ML 分类算法如何用来预测期货价格变动方向，并以这些预测为基础制定一个算法交易策略。所有细节将在第 15 章中解释，因此这个例子相当简洁。首先，导入数据，准备期货数据（定向滞后对数收益率数据）。

```
In [36]: import numpy as np
         import pandas as pd

In [37]: data = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                            index_col=0, parse_dates=True)
         data = pd.DataFrame(data['AAPL.O']) ❶
         data['Returns'] = np.log(data / data.shift()) ❷
         data.dropna(inplace=True)

In [38]: lags = 6

In [39]: cols = []
         for lag in range(1, lags + 1):
             col = 'lag_{}'.format(lag)
             data[col] = np.sign(data['Returns'].shift(lag)) ❸
             cols.append(col)
         data.dropna(inplace=True)
```

❶ 选择苹果公司（AAPL.O）历史日终数据。

❷ 计算整个历史数据内的对数收益率。

❸ 生成带有定向滞后对数收益率数据（+1 或者–1）的 DateFrame 列。

接下来，为支持向量机（SVM）算法、模型拟合和预测步骤实例化一个模型对象。图 1-2 说明，根据预测买进和卖出苹果公司股票的基于预测交易策略的表现胜过被动的基准股票投资方法。



图 1-2　基于 ML 的算法交易策略与被动基准投资的对比（苹果公司股票）

```
In [40]: from sklearn.svm import SVC

In [41]: model = SVC(gamma='auto') ❶

In [42]: model.fit(data[cols], np.sign(data['Returns'])) ❷
Out[42]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)

In [43]: data['Prediction'] = model.predict(data[cols]) ❸

In [44]: data['Strategy'] = data['Prediction'] * data['Returns'] ❹

In [45]: data[['Returns', 'Strategy']].cumsum().apply(np.exp).plot(
                 figsize=(10, 6)); ❺
```

❶ 实例化模型对象。

❷ 根据期货和标签数据（全部是定向的）拟合模型。

❸ 使用拟合模型创建预测（样本内），这同时是交易策略的头寸（买入或者卖出）。

❹ 根据预测值和基准对数收益率计算交易策略的对数收益率。

❺ 绘制基于 ML 交易策略表现与被动基准投资表现的对比曲线。

这里采用的简化方法没有考虑交易成本，也没有将数据集分为训练和测试子集。但是，这个例子说明，至少从技术意义上，将 ML 算法应用到金融数据有多么简单。在实践中，需要考虑一些重要的因素（参见 Lopez de Prado（2018））。

人工智能优先金融学

AI 将像对待其他领域一样，重塑金融行业。通过编程 API 功能可以取得大量金融数据，这成为了该领域的引擎。第 13 章中介绍 AI、ML 和 DL 基本方法，并在第 15 章和第 16 章中将其应用到算法交易中。不过，全面介绍人工智能优先金融学需要一本专门的图书。

作为数据驱动金融学的自然延伸，不管从研究还是从业者的角度，金融学中的人工智能当然也是一个令人痴迷和兴奋的领域。本书在不同背景下使用了多种 AI、ML 和 DL 方法，但总体的焦点与本书的副标题相符，仍是数据驱动金融学所需的 Python 基本技术与方法。不过，这些技术与方法对人工智能优先金融学同样重要。

1.5 结语

Python 作为一种语言，但更多的是作为生态系统，是金融业理想的技术框架。它的特

性中有许多好处，比如简洁的语法、高效的开发方法和原型化及生产的易用性等。利用 Python 大量的可用库和工具，我们似乎能够应付当今金融业中分析、数据量和频率、依从性及监管所引发的大部分问题。即使在较大型的金融机构中，它也具备提供单一、强大、一致性的框架，简化端到端开发和生产工作的潜力。

此外，Python 已经成为人工智能、特别是机器与深度学习工作者选择的编程语言。因此，Python 对于数据驱动金融学和人工智能优先金融学来说也是合适的语言，这两种当今的趋势将从根本上重塑金融学和金融行业。

1.6 延伸阅读

下面的图书更详细地介绍了本章浅释的多个方面（例如 Python 工具、衍生品研究、机器学习概述以及金融中的机器学习）。

- Hilpisch, Yves (2015). *Derivatives Analytics with Python*. Chichester, England: Wiley Finance.
- López de Prado, Marcos (2018). *Advances in Financial Machine Learning*. Hoboken, NJ: John Wiley & Sons.
- VanderPlas, Jake (2016). *Python Data Science Handbook*. Sebastopol, CA: O’Reilly.

至于算法交易，本书作者的公司提供了一系列在线培训项目，该计划聚焦于 Python 和其他工具，以及这个迅速成长的领域中所需的技术。

本章引用了如下信息来源。

- Ding, Cubillas (2010). “Optimizing the OTC Pricing and Valuation Infrastructure.” Celent.
- Lewis, Michael (2014). *Flash Boys*. New York: W. W. Norton & Company.
- Patterson, Scott (2010). *The Quants*. New York: Crown Business.

第 2 章

Python 基础架构

盖房子的时候，木料的选择是个问题。

> 木匠的目标本质上就是携带好的切割工具。在他有时间的时候，会将他的设备磨得很锋利。
>
> ——宫本武藏（五轮书）

对于 Python 新手来说，Python 的部署似乎很简单。丰富的可选安装库和程序包也很容易安装。首先，Python 不止一种，它有许多不同的“风味”，如 CPython、Jython、IronPython 和 PyPy。然后，Python 2.7 和 3.x 也是不同的世界。[1]

即便在你决定了使用哪一版本之后，部署也很困难，原因如下：

- 解释程序（标准 CPython 安装）只带有所谓的“标准库”（例如，包含了典型的数学函数）;
- 可选的 Python 软件包必须单独安装，它们有数百种之多；
- 由于依赖性和操作系统的特定需求，自行编译/构建这些非标准包可能很困难；
- 需要随时注意依赖性和版本一致性（也就是维护工作），这乏味且费时；
- 某些包的更新升级可能需要重新编译许多其他的软件包；
- 更新或者替换一个包，可能在其他（许多）地方造成麻烦。

幸运的是，我们可以求助于工具和策略。本章将介绍以下有助于 Python 部署的技术。

包管理器

pip 和 conda 等包管理器用于安装、更新和删除 Python 软件包；它们还有助于保持

1　本版基于 CPython 3.7（写作时最新的主版本），这是 Python 编程语言最受欢迎的原创版本。——原注

不同包的版本一致性。

虚拟环境管理器

virtualenv 或 conda 等虚拟环境管理器可以并行管理多种 Python 安装（例如，在单一机器上安装 Python 2.7 和 3.7，以便无风险地测试某个有趣的 Python 软件包的最新开发版本）。[1]

容器

Docker 容器代表着包含运行特定软件所需的全部系统部件（如代码、运行时库或系统工具）的完整文件系统。例如，你可以在运行 macOS 或者 Windows10 的一台机器上，在一个 Docker 容器里运行 Ubuntu 18.04 操作系统，并在其中安装 Python 3.7 和单独的 Python 代码。

云实例

为金融应用部署 Python 代码通常要求高可用性、安全性和高性能；这些需求一般只能利用专业计算及存储基础设施满足。目前，这种基础设施以情况良好的云实例形式存在，规模从小到大不等。与长期租赁的专用服务器相比，云实例（也就是虚拟服务器）的好处之一是，用户通常只需要按照实际使用的时间付费；另一个好处是，这种云实例在需要时一两分钟内就能投入使用，有助于敏捷开发并提高伸缩性。

本章的结构如下。

作为包管理器使用的 conda

本节介绍作为 Python 包管理器使用的 conda。

作为虚拟环境管理器使用的 conda

这一节专注于 conda 作为虚拟环境管理器的功能。

使用 Docker 容器

这一节简单概述了 Docker 容器化技术，聚焦于构建带有 Python 3.7 安装的 Ubuntu 容器。

使用云实例

本节介绍在云中部署 Python 和 Jupyter Notebook 的方法，后者是一个基于浏览器的、功能强大的 Python 部署工具套件。

本章的目标是在专业基础设施上建立一个合适的 Python 环境，包含最重要的工具以及数值、数据分析和可视化软件包（库）。此后，这一组合将作为实现和部署后续章节中

1　最新的 pipenv 项目结合了包管理器 pip 和虚拟环境管理器 virtualenv 的功能。——原注

Python 代码的支柱，不管这些代码是交互式的金融分析代码，还是以脚本和模块形式出现的。

2.1　作为包管理器使用的 conda

conda 可以单独安装，但是更有效的方法之一是通过 Miniconda 安装，这是一个包含 conda、并将 conda 作为包和虚拟环境管理器使用的最小化 Python 分发版本。

2.1.1　安装 Miniconda

Miniconda 可用于 Windows、macOS 和 Linux。你可以从 Miniconda 网页上下载不同的版本。以下假定安装的是 Python 3.7 64 位版本。本节的主要例子是基于 Ubuntu 的 Docker 容器中的一个会话，通过 wget 下载 Linux 64 位安装程序，然后安装 Miniconda。下面的代码应该可以在任何 Linux 或者 macOS 机器上正常工作（可能需要做少量修改）。

```
$ docker run -ti -h py4fi -p 11111:11111 ubuntu:latest /bin/bash

root@py4fi:/# apt-get update; apt-get upgrade -y
...
root@py4fi:/# apt-get install -y bzip2 gcc wget
...
root@py4fi:/# cd root
root@py4fi:~# wget \
> https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh \
> -O miniconda.sh
...
HTTP request sent, awaiting response... 200 OK
Length: 62574861 (60M) [application/x-sh]
Saving to: 'miniconda.sh'
miniconda.sh        100%[====================>] 59.68M 5.97MB/s in 11s

2018-09-15 09:44:28 (5.42 MB/s) - 'miniconda.sh' saved [62574861/62574861]

root@py4fi:~# bash miniconda.sh

Welcome to Miniconda3 4.5.11

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
```

按 Enter 键开始安装过程。阅读许可协议后，单击“yes”按钮同意条款。

```
...
Do you accept the license terms? [yes|no]
[no] >>> yes

Miniconda3 will now be installed into this location:
/root/miniconda3

 - Press ENTER to confirm the location
 - Press CTRL-C to abort the installation
 - Or specify a different location below

[/root/miniconda3] >>>
PREFIX=/root/miniconda3
installing: python-3.7. ...
...
installing: requests-2.19.1-py37_0 ...
installing: conda-4.5.11-py37_0 ...
installation finished.
```

同意许可条款并确认安装位置之后，你应该再次单击“yes”按钮，允许 Miniconda 将新的 Miniconda 安装位置附加到 PATH 环境变量中。

```
Do you wish the installer to prepend the Miniconda3 install location
to PATH in your /root/.bashrc ? [yes|no]
[no] >>> yes

Appending source /root/miniconda3/bin/activate to /root/.bashrc
A backup will be made to: /root/.bashrc-miniconda3.bak


For this change to become active, you have to open a new terminal.

Thank you for installing Miniconda3!
root@py4fi:~#
```

此后，你可能想要升级 conda 和 Python。[1]

```
root@py4fi:~# export PATH="/root/miniconda3/bin/:$PATH"
root@py4fi:~# conda update -y conda python
...
root@py4fi:~# echo ". /root/miniconda3/etc/profile.d/conda.sh" >> ~/.bashrc
root@py4fi:~# bash
```

在这一相当简单的安装过程之后，你就可以使用基本的 Python 安装和 conda 了。基本 Python 安装包含了一些很好的功能，如 SQLite3 数据库引擎。在对应的环境变量中附

1　Miniconda 安装程序通常不像 conda 和 Python 那样经常更新。——原注

加相关路径（正如前面所做的那样）之后，你可以试验能否从新的 Shell 实例中启动 Python：

```
root@py4fi:~# python
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
[GCC 7.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello Python for Finance World.')
Hello Python for Finance World.
>>> exit()
root@py4fi:~#
```

2.1.2 conda 基本操作

conda 可用于高效地处理 Python 软件包的安装、更新和删除。接下来概述一下主要功能。

安装 Python x.x 版本

```
conda install python=x.x
```

更新 Python

```
conda update python
```

安装软件包

```
conda install $PACKAGE_NAME
```

更新软件包

```
conda update $PACKAGE_NAME
```

删除软件包

```
conda remove $PACKAGE_NAME
```

更新 conda

```
conda update conda
```

搜索软件包

```
conda search $SEARCH_TERM
```

列出已安装软件包

```
conda list
```

考虑到这些功能，安装 NumPy——所谓“科学栈”中最重要的库之一——只需要一条

命令。在装有 Intel 处理器的机器上安装时，将自动安装 Intel 数学核心库（Intel Math Kernel Library，MKL），这个库不仅能加速 NumPy 的数值运算，而且对其他几个 Python 科学库也有作用。[1]

```
root@py4fi:~# conda install numpy
Solving environment: done

## Package Plan ##

  environment location: /root/miniconda3

  added / updated specs:
    numpy


The following packages will be downloaded:

    Package                    |            build
    ---------------------------|-----------------
    mkl-2019.0                 |              117       204.4 MB
    intel-openmp-2019.0        |              117         721 KB
    mkl_random-1.0.1           |   py37h4414c95_1         372 KB
    libgfortran-ng-7.3.0       |       hdf63c60_0         1.3 MB
    numpy-1.15.1               |   py37h1d66e8a_0          37 KB
    numpy-base-1.15.1          |   py37h81de0dd_0         4.2 MB
    blas-1.0                   |              mkl           6 KB
    mkl_fft-1.0.4              |   py37h4414c95_1         149 KB
------------------------------------------------------------
                                           Total:       211.1 MB

The following NEW packages will be INSTALLED:

blas:           1.0-mkl
intel-openmp:   2019.0-117
libgfortran-ng: 7.3.0-hdf63c60_0
mkl:            2019.0-117
mkl_fft:        1.0.4-py37h4414c95_1
mkl_random:     1.0.1-py37h4414c95_1
numpy:          1.15.1-py37h1d66e8a_0
numpy-base:     1.15.1-py37h81de0dd_0

Proceed ([y]/n)? y
```

1 安装元软件包 nomkl（例如使用命令 `conda install numpy nomkl`），可以避免自动安装和使用 mkl 及其他相关软件包。——原注

```
Downloading and Extracting Packages
mkl-2019.0          | 204.4 MB | ####################################### |
100%
...
numpy-1.15.1        | 37 KB    | ####################################### |
100%
numpy-base-1.15.1   | 4.2 MB   | ####################################### |
100%
...
root@py4fi:~#
```

也可以一次性安装多个软件包。`-y` 标志表示所有（可能的）问题都以 yes 回答。

```
root@py4fi:/# conda install -y ipython matplotlib pandas pytables scik
it-learn \
> scipy
...
pytables-3.4.4      | 1.5 MB   | ####################################### |
100%
kiwisolver-1.0.1    | 83 KB    | ####################################### |
100%
icu-58.2            | 22.5 MB  | ####################################### |
100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
root@py4fi:~#
```

安装过程结束后，除了标准库之外，一些重要的金融分析库也已经可用，包括：

IPython

改进的交互式 Python shell；

Matplotlib

Python 标准绘图库；

NumPy

用于高效处理数组；

pandas

用于管理表格数据，例如金融时间序列数据；

PyTables

Python HDF5 库封装器；

scikit-learn

　　机器学习和相关任务所用的软件包；

SciPy

　　一组科学类和函数（以依赖的形式安装）。

这就为数据分析、特别是金融分析提供了一个基本工具集。下面的例子使用 IPython 和 NumPy 提取了一组伪随机数。

```
root@py4fi:~# ipython
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import numpy as np

In [2]: np.random.seed(100)

In [3]: np.random.standard_normal((5, 4))
Out[3]:
array([[-1.74976547, 0.3426804 , 1.1530358 , -0.25243604],
       [ 0.98132079, 0.51421884, 0.22117967, -1.07004333],
       [-0.18949583, 0.25500144, -0.45802699, 0.43516349],
       [-0.58359505, 0.81684707, 0.67272081, -0.10441114],
       [-0.53128038, 1.02973269, -0.43813562, -1.11831825]])

In [4]: exit
root@py4fi:~#
```

执行 `conda list` 命令显示已安装的软件包。

```
root@py4fi:~# conda list
# packages in environment at /root/miniconda3:
#
# Name                    Version           Build  Channel
asn1crypto                0.24.0           py37_0
backcall                  0.1.0            py37_0
blas                      1.0                 mkl
blosc                     1.14.4       hdbcaa40_0
bzip2                     1.0.6        h14c3975_5
...
python                    3.7.0        hc3d631a_0
...
wheel                     0.31.1           py37_0
xz                        5.2.4        h14c3975_4
```

```
yaml                      0.1.7          had09818_2
zlib                      1.2.11         ha838bed_2
root@py4fi:~#
```

如果不再某个包需要，可以用 `conda remove` 将其高效删除。

```
root@py4fi:~# conda remove scikit-learn
Solving environment: done

## Package Plan ##

  environment location: /root/miniconda3
  removed specs:
    - scikit-learn


The following packages will be REMOVED:

    scikit-learn: 0.19.1-py37hedc7406_0

Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
root@py4fi:~#
```

conda 的包管理器功能相当实用。然而，在加上虚拟环境管理器后，你才能看出它的全部威力。

> **简易包管理**
>
> conda 作为包管理器，可以使 Python 软件包的安装、更新和删除变得很简单。没有必要自行构建和编译软件包——考虑到软件包指定的依赖列表和不同操作系统所需考虑的细节，这一步有时可能很棘手。

2.2 作为虚拟环境管理器的 conda

根据你所选择的安装程序版本，Miniconda 提供了默认的 Python 2.7 或者 3.7 安装。conda 的虚拟环境管理器可以允许不同的组合，例如，在 Python 3.7 默认安装的基础上增加一个完全独立的 Python 2.7.x 安装。为此，conda 提供了以下功能。

创建虚拟环境

```
conda create --name $ENVIRONMENT_NAME
```

激活环境

```
conda activate $ENVIRONMENT_NAME
```

停止环境

```
conda deactivate $ENVIRONMENT_NAME
```

删除环境

```
conda env remove --name $ENVIRONMENT_NAME
```

导出到环境文件

```
conda env export > $FILE_NAME
```

从一个文件中创建环境

```
conda env create -f $FILE_NAME
```

列出所有环境

```
conda info --envs
```

下面的代码是一个简单的示意：创建一个名为 py27 的环境，在其中安装 IPython，并执行一行 Python 2.7.x 代码。

```
root@py4fi:~# conda create --name py27 python=2.7
Solving environment: done

## Package Plan ##

  environment location: /root/miniconda3/envs/py27

  added / updated specs:
    - python=2.7

The following NEW packages will be INSTALLED:

    ca-certificates: 2018.03.07-0
...
    python:          2.7.15-h1571d57_0
...
    zlib:            1.2.11-ha838bed_2

Proceed ([y]/n)? y

Preparing transaction: done
```

```
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use:
# > conda activate py27
#
# To deactivate an active environment, use:
# > conda deactivate
#

root@py4fi:~#
```

注意激活环境之后提示符的变化（py27）。

```
root@py4fi:~# conda activate py27
(py27) root@py4fi:~# conda install ipython
Solving environment: done
...
Executing transaction: done
(py27) root@py4fi:~#
```

最后，以下代码允许你以 Python 2.7 语法使用 IPython。

```
(py27) root@py4fi:~# ipython
Python 2.7.15 |Anaconda, Inc.| (default, May 1 2018, 23:32:55)
Type "copyright", "credits" or "license" for more information.

IPython 5.8.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
Help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print "Hello Python for Finance World!"
Hello Python for Finance World!

In [2]: exit
(py27) root@py4fi:~#
```

如上例所示，将 conda 作为虚拟环境管理器使用，你可以同时安装不同的 Python 版本，还可以安装某些包的不同版本。默认 Python 的安装不受这一过程的影响，存在于同一台机器上的其他环境也是如此。`conda env list` 可显示所有可用环境。

```
(py27) root@py4fi:~# conda env list
# conda environments:
#
```

```
base                      /root/miniconda3
py27                   *  /root/miniconda3/envs/py27

(py27) root@py4fi:~#
```

有时候，你需要与其他人共享环境信息，或者在多台机器上使用环境信息。为此，可以用 `conda env export` 将已安装程序包列表导出到一个文件中。这只默认适用于使用相同操作系统的机器，因为结果 YAML 文件中指定构建版本，但可以将其删掉，只指定软件包版本。

```
(py27) root@py4fi:~# conda env export --no-builds > py27env.yml
(py27) root@py4fi:~# cat py27env.yml
name: py27
channels:
  - defaults
dependencies:
  - backports=1.0
...
  - python=2.7.15
...
  - zlib=1.2.11
prefix: /root/miniconda3/envs/py27

(py27) root@py4fi:~#
```

从技术上说，虚拟环境不过是一个特定的（子）文件夹结构，创建它们往往是为了进行一些快速测试[1]。在这种情况下，停止后的环境很容易用 `conda env remove` 删除。

```
(py27) root@py4fi:/# conda deactivate
root@py4fi:~# conda env remove -y --name py27

Remove all packages in environment /root/miniconda3/envs/py27:


## Package Plan ##

  environment location: /root/miniconda3/envs/py27


The following packages will be REMOVED:
```

1 在 Python 的官方文档中，可以找到如下解释：“Python‘虚拟环境’允许 Python 软件包安装于特定应用的独立位置中，而不是全局安装。”——原注

```
    backports:                         1.0-py27_1
...
    zlib:                              1.2.11-ha838bed_2

root@py4fi:~#
```

conda 的虚拟环境管理器的功能概述到此结束。

> **简易环境管理**
>
> conda 不仅有助于包管理，还是 Python 的虚拟环境管理器。它简化了不同 Python 环境的创建，允许在同一台机器上使用多个版本的 Python 和可选包，这些不同版本不会互相影响。conda 还可以导入环境信息，因此你可以轻松地在多台机器上复制环境，或者与他人共享。

2.3 使用 Docker 容器

Docker 容器已经征服了 IT 界。虽然这种技术仍然相对年轻，但已经为几乎所有类型软件应用的高效开发与部署确立了一个标杆。

在本书中，你可以将 Docker 容器看作一个独立（“容器化”）文件系统，它包含操作系统（例如 Ubuntu Server 18.04）、（Python）运行时库、其他系统与开发工具，以及需要的其他（Python）库和软件包。这样的一个 Docker 容器可以运行在 Windows 10 本地机器上，或者使用 Linux 操作系统的云实例上。

本节不打算介绍 Docker 容器所有激动人心的细节，而是简洁地说明 Docker 技术在 Python 开发领域所能发挥的能力。[1]

2.3.1 Docker 镜像和容器

不过，在说明之前，我们必须区分 Docker 的两个基本概念。首先是 Docker 镜像（Image），它可以比作 Python 的一个类。其次是 Docker 容器，它可以比作对应 Python 类的一个实例。[2]

你可以从 Docker 词汇表上找到镜像更为技术性的定义：

> Docker 镜像是容器的基础。镜像是根文件系统变化和用于容器运行时库内部的对应执行参数的一个有序集合。镜像通常包含一组相互堆叠的分层文件系统，它没有状态，也永远不会变化。

---

1 Docker 技术的全面介绍参见 *Matthias and Kane*（2015）。——原注

2 如果对这个术语的理解还不够清晰，不用担心，第 6 章将做进一步介绍。——原注

类似地，你可以在 Docker 词汇表上找到容器的定义，这使得与 Python 类和类实例的对比变得很清晰：

> 容器是 Docker 镜像的运行时实例。一个 Docker 实例包括：一个 Docker 镜像、一个执行环境和一个标准指令集。

根据操作系统的不同，Docker 的安装略有不同，这就是本节不做详述的原因。更多的信息和链接可以在 About Docker CE 页面上找到。

2.3.2　构建 Ubuntu 和 Python Docker 镜像

本节举例说明基于 Ubuntu 最新版本的 Docker 镜像的构建方法，镜像中还包括 Miniconda 和几个重要的 Python 软件包。此外，通过更新 Linux 软件包索引、在必要时升级软件包并安装某些附加的系统工具，我们可以进行一些 Linux 的“内务整理”工作。这方面的工作需要两个脚本，一个是完成所有 Linux 层面工作的 bash 脚本[1]。另一个是所谓的 *Dockerfile*，即控制镜像本身的构建过程。

例 2-1 中的 bash 安装脚本包括 3 个主要部分。第一部分处理 Linux“内务”。第二部分安装 Miniconda，第 3 部分安装 Python 可选程序包。更详细的注释请见脚本。

例 2–1　安装 Python 和可选包的脚本

```
#!/bin/bash
#
# Script to Install
# Linux System Tools and
# Basic Python Components
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
# GENERAL LINUX
apt-get update # updates the package index cache
apt-get upgrade -y # updates packages
# installs system tools
apt-get install -y bzip2 gcc git htop screen vim wget
apt-get upgrade -y bash # upgrades bash if necessary
apt-get clean # cleans up the package index cache

# INSTALL MINICONDA
# downloads Miniconda
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_
64.sh -O \ Miniconda.sh
```

1　Consult Robbins (2016)对 bash 脚本做了简洁的介绍和概述。——原注

```
bash Miniconda.sh -b # installs it
rm -rf Miniconda.sh # removes the installer
export PATH="/root/miniconda3/bin:$PATH" # prepends the new path

# INSTALL PYTHON LIBRARIES
conda update -y conda python # updates conda & Python (if required)
conda install -y pandas # installs pandas
conda install -y ipython # installs IPython shell
```

例 2-2 中的 Dockerfile 使用例 2-1 中的 bash 脚本，从而构建了一个新的 Docker 镜像。它的主要部分也有内嵌注释。

例 2-2　构建镜像的 Dockerfile

```
#
# Building a Docker Image with
# the Latest Ubuntu Version and
# Basic Python Install
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#

# latest Ubuntu version
FROM ubuntu:latest

# information about maintainer
MAINTAINER yves

# add the bash script
ADD install.sh /

# change rights for the script
RUN chmod u+x /install.sh

# run the bash script
RUN /install.sh

# prepend the new path
ENV PATH /root/miniconda3/bin:$PATH

# execute IPython when container is run
CMD ["ipython"]
```

如果这两个文件在一个文件夹里且安装了 Docker，那么构建新镜像就很简单了。这

里为镜像使用了 ubuntupython 标签，这个标签是引用镜像时必需的，例如在根据它运行容器时。

```
~/Docker$ docker build -t py4fi:basic .

...

Removing intermediate container 5fec0c9b2239
 ---> accee128d9e9
Step 6/7 : ENV PATH /root/miniconda3/bin:$PATH
 ---> Running in a2bb97686255
Removing intermediate container a2bb97686255
 ---> 73b00c215351
Step 7/7 : CMD ["ipython"]
 ---> Running in ec7acd90c991
Removing intermediate container ec7acd90c991
 ---> 6c36b9117cd2
Successfully built 6c36b9117cd2
Successfully tagged py4fi:basic
~/Docker$
```

现有 Docker 镜像可以用 docker images 列出。新镜像出现在列表的开头：

```
(py4fi) ~/Docker$ docker images
REPOSITORY       TAG            IMAGE ID      CREATED             SIZE
py4fi            basic          f789dd230d6f  About a minute ago  1.79GB
ubuntu           latest         cd6d8154f1e1  9 days ago          84.1MB
(py4fi) ~/Docker$
```

成功地构建 py4fi:basic 之后，你就可以用 docker run 运行对应的 Docker 容器。参数组合-ti 是在 Docker 容器内部运行交互过程（如 Shell 过程）时所需的（参见 docker run 参考页面）：

```
~/Docker$ docker run -ti py4fi:basic
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import numpy as np

In [2]: a = np.random.standard_normal((5, 3))

In [3]: import pandas as pd

In [4]: df = pd.DataFrame(a, columns=['a', 'b', 'c'])

In [5]: df
```

```
Out[5]:
          a         b        c
0 -1.412661 -0.881592 1.704623
1 -1.294977  0.546676 1.027046
2 1.156361   1.979057 0.989772
3 0.546736  -0.479821 0.693907
4 -1.972943 -0.193964 0.769500

In [6]:
```

退出 IPython 也将退出该容器，因为 IPython 是容器中运行的唯一应用程序。不过，你可以输入 Ctrl-P、Ctrl-Q 来卸下容器。

卸下容器之后，`docker ps` 命令仍会显示运行中的容器（以及当前运行的其他任何容器）：

```
~/Docker$ docker ps
CONTAINER ID  IMAGE         COMMAND     CREATED            STATUS
e815df8f0f4d  py4fi:basic   "ipython"   About a minute ago Up About a minute
4518917de7dc  ubuntu:latest "/bin/bash" About an hour ago  Up About an hour
d081b5c7add0  ubuntu:latest "/bin/bash" 21 hours ago       Up 21 hours
~/Docker$
```

Docker 容器的连接可用 `docker attach $CONTAINER_ID` 命令完成（注意，`$CONTAINER_ID` 只需要几个字符就够了）：

```
~/Docker$ docker attach e815d

In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
a    5 non-null float64
b    5 non-null float64
c    5 non-null float64
dtypes: float64(3)
memory usage: 200.0 bytes

In [7]: exit
~/Docker$
```

`exit` 命令会终止 IPython 并停止 Docker 容器。容器可用 `docker rm` 删除：

```
~/Docker$ docker rm e815d
e815d
~/Docker$
```

类似地，如果不再需要 Docker 容器 `py4fi:basic`，可以通过 `docker rmi` 删除。虽

然容器相对轻量，但单独容器仍可能消耗相当一部分存储容量。以 `py4fi:basic` 为例，其大小接近 2GB。这也就是你可能想要定期清理 Docker 镜像列表的原因：

```
~/Docker$ docker rmi 6c36b9117cd2
```

当然，关于 Docker 容器及其在某些应用场景中的好处，还有许多好的讲解。但对于本书，你只需要知道，容器为部署 Python、在完全独立（容器化）的环境中完成 Python 开发和为交付算法交易代码提供了一种现代化方法，就足够了。

Docker 容器的好处

如果你还没有使用 Docker 容器，应该考虑试试它。它们为 Python 部署与开发工作提供了许多好处，不仅在本地工作时如此，在使用远程云实例与服务器部署算法交易代码时更具优势。

2.4　使用云实例

本节介绍在 DigitalOcean 云实例上建立全套 Python 基础架构的方法。其他的云提供商还有很多，如业界领先的 Amazon Web Services。但是 Digital Ocean 以简易而闻名，对于小型云实例（称作水滴（Droplet））来说，价格也相对低。研究和开发通常使用最小的 Droplet 就足够了，其费用只有每月 5 美元（或者每小时 0.007 美元）。收费按小时计算，因此人们可以轻松地使用一个 Droplet 两个小时，然后将其删除，这只需要支付 0.014 美元。[1]

本节的目标是在 DigitalOcean 上建立一个 Droplet，它包含 Python 3.7 安装包和典型的软件包（例如 NumPy、pandas），并与一个密码保护、用安全套接字层（SSL）加密的 Jupyter Notebook 服务器安装相结合。这个服务器安装将提供 3 个可通过常规浏览器使用的重要工具。

Jupyter Notebook

流行的交互开发环境，其特点是可以选择不同语言内核（例如，Python、R 和 Julia）。

终端

　　可通过浏览器访问的系统 Shell 实现，能完成所有典型系统管理任务、对 Vim 和 git 等实用工具的使用。

编辑器

　　基于浏览器的文件编辑器，可以对许多不同编程语言及文件类型的语法高亮显示，还具有典型的文本/代码编辑功能。

1　通过推荐链接注册的新用户可以得到 Digital Ocean 的 10 美元信用额度。——原注

在 Droplet 上安装 Jupyter Notebook 之后，你可以通过浏览器进行 Python 开发和部署，而不需要通过安全外壳（SSH）访问、登录云实例。

要实现本节的目标，需要几个文件。

服务器设置脚本

　　这个脚本协调所有必要步骤，例如将其他文件复制到 Droplet 中，并在其上运行。

Python 和 Jupyter 安装脚本

　　这个脚本用于安装 Python、附加软件包和 Jupyter Notebook，并启动 Jupyter Notebook 服务器。

Jupyter Notebook 配置文件

　　这个文件用于配置 Jupyter Notebook 服务器，例如有关密码保护的细节。

RSA 公钥与私钥文件

　　这两个文件是 Jupyter Notebook 服务器 SSL 加密时所需的。

下面的小节以相反的次序来处理上述文件。

2.4.1　RSA 公钥和私钥

为了通过任意浏览器建立与 Jupyter Notebook 服务器的安全连接，我们需要一个包含 RSA 公钥和私钥的 SSL 证书。一般来说，这样的证书来自所谓的证书颁发机构（CA）。但是，在本书中，自生成的证书就“足够好”了。[1]生成 RSA 密钥对的流行工具之一是 OpenSSL。下面简短的交互会话说明如何生成用于 Jupyter Notebook 服务器的证书（在提示符后面插入你自己的国家名称和其他字段）：

```
~/cloud$ openssl req -x509 -nodes -days 365 -newkey \
> rsa:1024 -out cert.pem -keyout cert.key
Generating a 1024 bit RSA private key
..++++++
.......++++++
writing new private key to 'cert.key'
```

系统将要求你输入加进证书请求中的信息。你将输入所谓的可识别名称（DN）。需要输入的字段有多个，但你可以将某些留空，其他一些则用默认值。如果输入“.”，字段将留空：

```
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:Saarland
Locality Name (eg, city) []:Voelklingen
```

1　如果使用自生成证书，你可能需要在浏览器提示时添加一个安全例外。——原注

```
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TPQ GmbH
Organizational Unit Name (eg, section) []:Python for Finance
Common Name (e.g. server FQDN or YOUR name) []:Jupyter
Email Address []:team@tpq.io
~/cloud$ ls
cert.key cert.pem
~/cloud$
```

cert.key 和 cert.pem 这两个文件必须复制到 Droplet 中，由 Jupyter Notebook 配置文件引用。下面介绍配置文件。

2.4.2 Jupyter Notebook 配置文件

你可以按照 Jupyter Notebook 文档中的解释来部署一个公共 Jupyter Notebook 服务器。除了其他特性之外，Jupyter Notebook 还可以加上密码保护。为此，`notebook.auth` 子软件包提供了一个密码散列代码生成函数 `passwd()`。以下代码生成密码为 jupyter 的散列代码：

```
~/cloud$ ipython
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from notebook.auth import passwd

In [2]: passwd('jupyter')
Out[2]: 'sha1:d4d34232ac3a:55ea0ffd78cc3299e3e5e6ecc0d36be0935d424b'

In [3]: exit
```

这个散列代码必须放在例 2-3 提供的 Jupyter Notebook 配置文件中。该配置文件假定 RSA 公钥文件已经复制到 Droplet 上的/root/.jupyter/文件夹中。

例 2-3　Jupyter Notebook 配置文件

```
#
# Jupyter Notebook Configuration File
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
# SSL ENCRYPTION
# replace the following filenames (and files used) with your choice/files
c.NotebookApp.certfile = u'/root/.jupyter/cert.pem'
c.NotebookApp.keyfile = u'/root/.jupyter/cert.key'
```

```
# IP ADDRESS AND PORT
# set ip to '*' to bind on all IP addresses of the cloud instance
c.NotebookApp.ip = '*'
# it is a good idea to set a known, fixed default port for server access
c.NotebookApp.port = 8888

# PASSWORD PROTECTION
# here: 'jupyter' as password
# replace the hash code with the one for your strong password

c.NotebookApp.password = 'sha1:d4d34232ac3a:55ea0ffd78cc3299e3e5e6ecc0
d36be0935d424b'
# NO BROWSER OPTION
# prevent Jupyter from trying to open a browser
c.NotebookApp.open_browser = False
```

Jupyter 与安全性

在云端部署 Jupyter Notebook 会引起一些安全问题，因为这是一个通过浏览器访问的完整开发环境。因此，最重要的是使用 Jupyter Notebook 服务器提供的安全措施，例如密码保护和 SSL 加密。但这只是开始：建议根据在云实例上完成的工作采取进一步的安全措施。

下一步是确保将 Python 和 Jupyter Notebook 安装在 Droplet 上。

2.4.3 Python 和 Jupyter Notebook 安装脚本

安装 Python 和 Jupyter Notebook 的 bash 脚本与 2.3 节中通过 Docker 容器里的 Miniconda 安装 Python 的脚本类似。但是，例 2-4 中的脚本还需要启动 Jupyter Notebook 服务器。所有的重要部分和代码行都有内嵌注释。

例 2-4 安装 Python 和运行 Jupyter Notebook 服务器的 bash 脚本

```
#!/bin/bash
#
# Script to Install
# Linux System Tools,
# Basic Python Packages and
# Jupyter Notebook Server
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
# GENERAL LINUX
```

```
apt-get update # updates the package index cache
apt-get upgrade -y # updates packages
apt-get install -y bzip2 gcc git htop screen vim wget # installs system tools
apt-get upgrade -y bash # upgrades bash if necessary
apt-get clean # cleans up the package index cache

# INSTALLING MINICONDA
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_
64.sh -O \ Miniconda.sh
bash Miniconda.sh -b # installs Miniconda
rm Miniconda.sh # removes the installer
# prepends the new path for current session
export PATH="/root/miniconda3/bin:$PATH"
# prepends the new path in the shell configuration
echo ". /root/miniconda3/etc/profile.d/conda.sh" >> ~/.bashrc
echo "conda activate" >> ~/.bashrc

# INSTALLING PYTHON LIBRARIES
# More packages can/must be added
# depending on the use case.
conda update -y conda # updates conda if required
conda create -y -n py4fi python=3.7 # creates an environment
source activate py4fi # activates the new environment
conda install -y jupyter # interactive data analytics in the browser
conda install -y pytables # wrapper for HDF5 binary storage
conda install -y pandas # data analysis package
conda install -y matplotlib # standard plotting library
conda install -y scikit-learn # machine learning library
conda install -y openpyxl # library for Excel interaction
conda install -y pyyaml # library to manage YAML files

pip install --upgrade pip # upgrades the package manager
pip install cufflinks # combining plotly with pandas

# COPYING FILES AND CREATING DIRECTORIES
mkdir /root/.jupyter
mv /root/jupyter_notebook_config.py /root/.jupyter/
mv /root/cert.* /root/.jupyter
mkdir /root/notebook
cd /root/notebook

# STARTING JUPYTER NOTEBOOK
jupyter notebook --allow-root
```

```
# STARTING JUPYTER NOTEBOOK
# as background process:
# jupyter notebook --allow-root &
```

这个脚本必须复制到 Droplet，并由下一小节介绍的编配脚本启动。

2.4.4　协调 Droplet 设置的脚本

设置 Droplet 的第二个 bash 脚本最短（例 2-5）。它主要的作用是将所有其他文件复制到 Droplet 中，Droplet 的 IP 地址是脚本的一个参数。脚本的最后一行用于启动 install.sh 脚本，后者安装完成后启动 Jupyter Notebook 服务器。

例 2-5　设置 Droplet 的 bash 脚本

```
#!/bin/bash
#
# Setting up a DigitalOcean Droplet
# with Basic Python Stack
# and Jupyter Notebook
#
# Python for Finance, 2nd ed.
# (c) Dr Yves J Hilpisch
#

# IP ADDRESS FROM PARAMETER
MASTER_IP=$1

# COPYING THE FILES
scp install.sh root@${MASTER_IP}:
scp cert.* jupyter_notebook_config.py root@${MASTER_IP}:

# EXECUTING THE INSTALLATION SCRIPT
ssh root@${MASTER_IP} bash /root/install.sh
```

现在，设置代码的所有条件都已经具备。在 DigitalOcean 上，用如下选项创建一个新的 Droplet。

操作系统

Ubuntu 18.10 x64（本书写作时最新的版本）。

大小

1 个内核、1GB 内存、25GB SSD（最小的 Droplet）。

数据中心所在地区

法兰克福（因为作者住在德国）。

SSH 密钥

添加一个（新）SSH 密钥，用于无密码登录。[1]

Droplet 名称

你可以使用预先指定的名称，或者选择 `py4fi` 等名称。

单击 Create（创建）按钮启动 Droplet 创建过程，该过程通常需要花费一分钟。设置过程的主要结果是一个 IP 地址，如果你选择法兰克福作为数据中心位置，这个地址可能是 46.101.156.199。现在，设置 Droplet 只需要以下简单命令：

```
(py3) ~/cloud$ bash setup.sh 46.101.156.199
```

后续的过程可能需要几分钟。结束时，Jupyter Notebook 服务器将显示以下信息：

```
The Jupyter Notebook is running at: https://[all ip addresses on your
system]:8888/
```

在任何现代浏览器中，访问以下地址都可以访问运行中的 Jupyter Notebook 服务器（注意 https 协议）：

```
https://46.101.156.199:8888
```

服务器可能要求你添加一个安全例外，添加后应该会出现 Jupyter Notebook 登录屏幕，提示输入密码（我们的例子中是 jupyter）。现在，你已经做好准备，在浏览器中通过 Jupyter Notebook、通过终端窗口中的 IPython 或者文本文件编辑器开始 Python 开发。其他文件管理功能（如文件上传、删除和文件夹创建）也已经具备。

云的好处

DigitalOcean 等公司提供的云实例和 Jupyter Notebook 组合起来很强大，使 Python 开发人员和“宽客”可以使用专业的计算和存储基础设施。专业的云和数据中心提供商确保你的（虚拟）机器的物理安全和高可用性。使用云实例还可以将研究和开发阶段的成本保持在较低的水平，因为收费通常按照使用的小时数计算，不需要签订长期的协议。

2.5 结语

Python 是本书选用的编程语言和技术平台，并且它被几乎所有领先的金融机构采用。不过，Python 开发可能很难，有时候甚至是乏味且令人头疼的。幸运的是，近年来已

1 如果需要帮助，访问“How to Add SSH Keys to Droplets”或者“How to Create SSH Keys with PuTTY on Windows”网页。——原注

经出现了许多协助解决部署问题的技术。开源项目 conda 可以帮助管理 Python 软件包和虚拟环境。Docker 容器更进一步，可以轻松地在技术上隔离的“沙箱”（容器）中创建完整的文件系统和运行时环境。DigitalOcean 等云提供商更能在几分钟之内，提供专业管理的安全数据中心内的计算及存储能力，按照小时收费。这些技术与 Python 3.7 安装和安全的 Jupyter Notebook 服务器安装相结合，为 Python 金融项目的开发和部署提供了专业化的环境。

2.6 延伸阅读

Python 软件包管理可以参考如下资源：

- pip 包管理器主页；
- conda 包管理器主页；
- Python 安装软件包页面。

虚拟环境管理可以参考如下资源：

- virtualenv 环境管理器页面；
- conda 环境管理页面；
- pipenv 包与环境管理器。

下面的资源提供关于 Docker 容器的信息：

- Docker 主页；
- Matthias, Karl, and Sean Kane (2015). *Docker: Up and Running*. Sebastopol, CA: O’Reilly。

bash 脚本语言的简介与概述参见：

- Robbins, Arnold (2016). *Bash Pocket Reference*. Sebastopol, CA: O’Reilly。

Jupyter Notebook 文档解释了安全运行公共 Jupyter Notebook 服务器的方法。JupyterHub 是管理 Jupyter Notebook 服务器的多个用户的一个中心。

通过推荐人的链接注册 DigitalOcean，并在新账户中得到 10 美元的起始余额。这可以支付最小 Droplet 两个月的使用费。

第 2 部分

掌握基础知识

本书的这一部分关注 Python 编程基础，所介绍的主题是后续部分和使用 Python 的根本。

该部分的各章按照特定主题组织，读者可通过这些主题查找与感兴趣主题相关的示例和细节：

- 第 3 章专注于 Python 数据类型和结构；
- 第 4 章介绍 NumPy 及其 `ndarray` 类的相关知识；
- 第 5 章介绍 pandas 及其 `DataFrame` 类的相关知识；
- 第 6 章讨论 Python 面向对象编程（OOP）。

第 3 章
数据类型与结构

拙劣的程序员担心代码。好的程序员担心数据结构及它们的关系。

——林纳斯 · 托瓦兹

本章介绍 Python 的基本数据类型和数据结构，它们按照如下方式组织。

基本数据类型

3.1 节介绍整数、浮点数和字符串等基本数据类型。

基本数据结构

3.2 节介绍 Python 的基本数据结构（例如 List 对象）并阐述控制结构、函数式编程范型以及匿名函数。

本章的主旨是提供 Python 数据类型和结构细节的一般介绍。如果你有其他编程语言（如 C 或者 Matlab）的背景，应该能够轻松地掌握 Python 使用方法带来的差异。本章介绍的主题是后续章节的重要基础，涵盖了以下数据类型与结构。

对象类型	含义	用途
`int`	整数值	自然数
`float`	浮点数	实数
`bool`	布尔值	“真”或“假”
`str`	字符串对象	字符、单词、文本
`tuple`	不可变容器	固定对象集、记录
`list`	可变容器	变化的对象集
`dict`	可变容器	键-值存储
`set`	可变容器	独特对象集

3.1 基本数据类型

Python 是一种动态类型的语言，这意味着，Python 解释程序在运行时推断对象的类型。C 等编译语言通常是静态类型语言，在这类语言中，对象类型必须在编译之前与对象绑定[1]。

3.1.1 整数

最基本的数据类型是整数 `int`：

```
In [1]: a = 10
        type(a)
Out[1]: int
```

内建函数 `type` 为所有使用标准和内建类型的对象、新创建的类及对象提供类型信息。在后一种情况下，提供的信息取决于程序员存储在类中的描述。有一种说法："在 Python 中，一切都是对象。"这意味着，即使我们刚刚定义的简单对象 int 也有内建方法。例如，你可以调用 `bit_length` 方法，获得表现整数对象所需的位数：

```
In [2]: a.bit_length()
Out[2]: 4
```

你将会看到，我们为该对象所赋的整数值越大，需要的位数就越多：

```
In [3]: a = 100000
        a.bit_length()
Out[3]: 17
```

一般来说，这类方法很多，很难记住所有类和对象的所有方法。高级 Python 环境（如 IPython）提供 Tab 键完成功能，以显示对象连接的所有方法。你只需要输入对象名和一个点（如 `a.`）然后按下 Tab 键，就会提供一组可以在该对象上调用的方法。也可以使用 Python 内建函数 dir 显示任何对象属性和方法的完整列表。

Python 的特殊性之一是整数可以任意大。例如，考虑天文数字 10^{100}。Python 处理这样大的数毫无问题，这从技术上说是一个很"长"的对象：

```
In [4]: googol = 10 ** 100
        googol
Out[4]: 1000000000000000000000000000000000000000000000000000000000000
        0000000000000000000000000000000000000000
```

1 Cython 库为 Python 引入了可与 C 相比的静态类型和编译特性。实际上，Cython 是结合 Python 和 C 的完备"混血"编程语言。——原注

```
In [5]: googol.bit_length()
Out[5]: 333
```

大整数

Python 整数可以为任意大。解释程序简单地使用所需的位/字节数就可表现数值。

整数的算术运算也很容易实现：

```
In [6]: 1 + 4
Out[6]: 5

In [7]: 1 / 4
Out[7]: 0.25

In [8]: type(1 / 4)
Out[8]: float
```

3.1.2 浮点数

为了让上一个表达式返回所需的值——0.25[1]，我们必须在浮点（float）对象上操作，这很自然地把我们带到下一种基本数据类型。在整数值上加一个点比如 1.或者 1.0，会导致 Python 将该对象解释为浮点数。涉及浮点数的表达式通常也必须返回一个浮点对象[2]：

```
In [9]: 1.6 / 4
Out[9]: 0.4
In [10]: type (1.6 / 4)
Out[10]: float
```

浮点数与通常不精确的实数计算机表现形式关系更大，取决于所采用的具体技术方法。为了说明其中的含义，我们定义另一个浮点对象 b，这种浮点对象在内部总是只表现为某种精度。将 b 加上 0.1 时，这一点变得很明显：

```
In [11]: b = 0.35
         type(b)
Out[11]: float

In [12]: b + 0.1
Out[12]: 0.44999999999999996
```

1 与版本中默认采用向下取整除法的 Python 2.x 不同，Python 3.x 中向下取整除法用 3//4 的形式实现，结果为 0。——原注

2 这里和后面的讨论中，浮点数、浮点对象等经常互换使用，每个浮点数都是对象。对于其他对象类型也是如此。——原注

出现以上结果的原因是浮点数在计算机内部以二进制形式表示；也就是说，十进制数 n（$0<n<1$）表现为如下形式：$n=\frac{x}{2}+\frac{y}{4}+\frac{z}{8}+\cdots$ 对于某些浮点数，二进制表现形式可能包含大量元素甚至是一个无限系列。但是，考虑到用于表达这种数值的位数是固定的——也就是表现系列中的项目固定，因此结果是不精确的。其他数值可以完整表现，因此在可用位数有限时也可以精确地被存储。考虑如下例子：

```
In [13]: c = 0.5
         c.as_integer_ratio()
Out[13]: (1, 2)
```

0.5 可以精确保存，因为它具备精确（有限）的二进制表示：0.5=1/2。但是，b=0.35 和预期的实数 0.35=7/20 不同：

```
In [14]: b.as_integer_ratio()
Out[14]: (3152519739159347, 9007199254740992)
```

精度取决于表示数值所用的位数。一般来说，Python 运行的所有平台使用 IEEE 754 双精度标准（64 位）作为内部表示，这相当于 15 位数字的相对精度。

由于这个主题在金融应用领域中很重要，所以有时候必须确保数值的精确（至少尽可能达到最佳）。例如，在加总一组数量很多的数值时，这个问题就可能很重要。在这种情况下，某个种类或者幅度的表示误差可能汇聚起来，从而造成和基准值的显著偏差。

decimal 模块为浮点数提供了任意精确度的对象，以及使用这些数值时处理精度问题的多个选项：

```
In [15]: import decimal
         from decimal import Decimal

In [16]: decimal.getcontext()
Out[16]: Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999,
         Emax=999999, capitals=1, clamp=0, flags=[], traps=
         [InvalidOperation, DivisionByZero, Overflow])

In [17]: d = Decimal(1) / Decimal (11)
         d
Out[17]: Decimal('0.09090909090909090909090909091')
```

你可以改变 `Context` 对象的各个属性值，从而改变表示的精度：

```
In [18]: decimal.getcontext().prec = 4❶

In [19]: e = Decimal(1) / Decimal (11)
         e
Out[19]: Decimal('0.09091')
```

```
In [20]: decimal.getcontext().prec = 50❷

In [21]: f = Decimal(1) / Decimal (11)
         f
Out[21]: Decimal('0.090909090909090909090909090909090909090909090909091')
```

❶ 精度低于默认值。

❷ 精度高于默认值。

如果有必要，精度可以这样按照具体的问题进行调整，人们可以使用不同精度的浮点对象进行运算：

```
In [22]: g = d + e + f
         g
Out[22]: Decimal('0.27272818181818181818181818181909090909090909090909')
```

> **任意精度浮点数**
>
> decimal 模块可提供任意精度浮点数对象。在金融领域中，确保高精度、超出 64 位双精度标准有时是必要的。

3.1.3 布尔值

在编程中，比较或者逻辑表达式（如 4 > 3, 4.5 <= 3.25 或 (4 > 3) and (3 > 2)）得到的输出为 `True` 或者 `False`，这是两个重要的 Python 关键字。其他重要的关键字有 `def`、`for` 和 `if` 等。Python 的关键字列表可以在 `keyword` 模块中找到：

```
In [23]: import keyword

In [24]: keyword.kwlist
Out[24]: ['False',
          'None',
          'True',
          'and',
          'as',
          'assert',
          'async',
          'await',
          'break',
          'class',
          'continue',
          'def',
          'del',
          'elif',
```

```
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

`True` 和 `False` 的数据类型为 `bool`，表示“布尔值”（Boolean Value）。将 Python 的比较运算符应用到相同操作数时结果为布尔对象：

```
In [25]: 4 > 3❶
Out[25]: True

In [26]: type(4 > 3)
Out[26]: bool

In [27]: type(False)
Out[27]: bool

In [28]: 4 >= 3❷
Out[28]: True

In [29]: 4 < 3❸
Out[29]: False

In [30]: 4 <= 3❹
Out[30]: False

In [31]: 4 == 3❺
Out[31]: False
```

```
In [32]: 4 != 3❻
Out[32]: True
```

❶ 大于。

❷ 大于或等于。

❸ 小于。

❹ 小于或等于。

❺ 等于。

❻ 不等于。

逻辑运算往往应用到布尔对象上，然后得出另一个布尔对象：

```
In [33]: True and True
Out[33]: True

In [34]: True and False
Out[34]: False

In [35]: False and False
Out[35]: False

In [36]: True or True
Out[36]: True

In [37]: True or False
Out[37]: True

In [38]: False or False
Out[38]: False

In [39]: not True
Out[39]: False

In [40]: not False
Out[40]: True
```

当然，两类运算符往往组合使用：

```
In [41]: (4 > 3) and (2 > 3)
Out[41]: False

In [42]: (4 == 3) or (2 != 3)
```

```
Out[42]: True

In [43]: not (4 != 4)
Out[43]: True

In [44]: (not (4 != 4)) and (2 == 3)
Out[44]: False
```

布尔值的重要应用领域之一是通过其他 Python 关键字（如 `if` 或 `while`，本章后面将有更多示例）控制代码流程：

```
In [45]: if 4 > 3: ❶
              print('condition true') ❷
         condition true
In [46]: i = 0 ❸
         while i < 4: ❹
             print('condition true, i = ', i) ❺
             i += 1 ❻
         condition true, i = 0
         condition true, i = 1
         condition true, i = 2
         condition true, i = 3
```

❶ 如果条件为真，执行以下代码。

❷ 如果条件为真，执行这段代码。

❸ 将参数 `i` 初始化为 0。

❹ 只要条件为 0，就执行并重复以下代码。

❺ 打印文本和参数 `i` 的值。

❻ 将参数增加 1；`i += 1` 与 `i = i + 1` 等价。

从数值上讲，Python 为 `False` 赋值 0，为 `True` 赋值 1。当通过 `bool()` 函数将数值转换成 `bool` 对象时，0 值转换成 `False`，其他所有值都转换为 `True`：

```
In [47]: int(True)
Out[47]: 1

In [48]: int(False)
Out[48]: 0

In [49]: float(True)
Out[49]: 1.0

In [50]: float(False)
```

```
Out[50]: 0.0

In [51]: bool(0)
Out[51]: False

In [52]: bool(0.0)
Out[52]: False

In [53]: bool(1)
Out[53]: True

In [54]: bool(10.5)
Out[54]: True

In [55]: bool(-2)
Out[55]: True
```

3.1.4 字符串

既然我们已经可以表示自然和浮点数，现在就可以转向文本了。Python 中表示文本的基本数据类型是字符串——`str`，字符串对象有一些真正实用的内建方法。实际上，Python 通常被视为处理任何类型、任何尺寸文本文件的很好选择。字符串对象通常由单引号或者双引号定义或者使用 `str()`函数转换而来（也就是使用对象的标准或者用户定义的字符串表示）：

```
In [56]: t = 'this is a string object'
```

举个内建方法的例子，你可以将对象中的第一个词改为首字母大写：

```
In [57]: t.capitalize()
Out[57]: 'This is a string object'
```

也可以将字符串拆分成单个词，以获得包含所有单词的列表对象：

```
In [58]: t.split()
Out[58]: ['this', 'is', 'a', 'string', 'object']
```

你还可以搜索一个单词，如果搜索成功，就可以得到该词第一个字母的位置（即索引值）：

```
In [59]: t.find('string')
Out[59]: 10
```

如果这个单词不在字符串对象中，该方法返回-1：

```
In [60]: t.find('Python')
Out[60]: -1
```

替换字符串中的字符是典型的任务，可以用 replace()方法简单地完成：

```
In [61]: t.replace(' ', '|')
Out[61]: 'this|is|a|string|object'
```

字符串“剥离”（stripping）操作——删除某些前导或者后缀字符也是必要的：

```
In [62]: 'http://www.python.org'.strip('htp:/')
Out[62]: 'www.python.org'
```

表 3-1 列出了字符串对象的一些实用方法。

表 3-1　精选字符串方法

方　　法	参　　数	返回/结果
capitalize	()	复制字符串，将第一个字符改成大写
count	sub[, start[, end]])	计算子字符串出现的次数
decode	([encoding[, errors]])	用 encoding 指定的编码方式（例如 UTF-8）解码字符串
encode	([encoding[, errors]])	字符串编码形式
find	(sub[, start[, end]])	找到的子字符串（最低）索引
join	(seq)	连接 seq 序列中的字符串
replace	(old, new[, count])	用 new 替换前 count 个 old
split	([sep[, maxsplit]])	字符串中的单词列表，以 sep 作为分隔符
splitlines	([keepends])	如果 keepends 为真，返回以行结束符/换行符分隔的行
strip	(chars)	从字符串首/尾删除 chars 中的字符
upper	()	复制字符串，所有字母改为大写

Unicode 字符串

Python 2.7（本书第 1 版使用）到 Python 3.7（本书第 2 版使用）的根本变化之一是字符串对象的编码与解码方法，以及 Unicode 的引入。本章不介绍这方面的细节；本书主要处理数值数据和包含英语单词的标准字符串，这种忽略似乎是合理的。

3.1.5　题外话：打印和字符串替换

打印字符串对象或者其他 Python 对象的字符串表示，通常由 print()函数完成：

```
In [63]: print('Python for Finance') ❶
```

```
         Python for Finance

In [64]: print(t) ❷
         this is a string object

In [65]: i = 0
         while i < 4:
             print(i) ❸
             i += 1
         0
         1
         2
         3
In [66]: i = 0
         while i < 4:
             print(i, end='|') ❹
             i += 1
         0|1|2|3|
```

❶ 打印一个 `str` 对象。

❷ 打印变量名引用的字符串对象。

❸ 打印整数对象的字符串表示。

❹ 指定打印的结束符，默认为前面见到的换行符（`\n`）。

Python 提供了强大的字符串替换运算。我们可以采用旧方法即通过%字符来实现，也可以采用通过花括号（{}）和 `format()`函数的新方法实现。两种方法在实践中都适用。本节无法提供所有选项的说明，但以下代码片段可以展示一些重要的选项。首先是旧方法：

```
In [67]: 'this is an integer %d' % 15 ❶
Out[67]: 'this is an integer 15'

In [68]: 'this is an integer %4d' % 15 ❷
Out[68]: 'this is an integer 15'

In [69]: 'this is an integer %04d' % 15 ❸
Out[69]: 'this is an integer 0015'

In [70]: 'this is a float %f' % 15.3456 ❹
Out[70]: 'this is a float 15.345600'

In [71]: 'this is a float %.2f' % 15.3456 ❺
Out[71]: 'this is a float 15.35'

In [72]: 'this is a float %8f' % 15.3456 ❻
```

```
Out[72]: 'this is a float 15.345600'

In [73]: 'this is a float %8.2f' % 15.3456 ❼
Out[73]: 'this is a float    15.35'

In [74]: 'this is a float %08.2f' % 15.3456 ❽
Out[74]: 'this is a float 00015.35'

In [75]: 'this is a string %s' % 'Python' ❾
Out[75]: 'this is a string Python'

In [76]: 'this is a string %10s' % 'Python' ❿
Out[76]: 'this is a string     Python'
```

❶ 整数对象替换。

❷ 固定数量的字符

❸ 必要时加上前导 0。

❹ 浮点数对象替换。

❺ 固定小数位数。

❻ 固定数量字符（以及填充的小数位数）。

❼ 固定数量字符和小数位数等。

❽ 必要时加上前导 0。

❾ 字符串对象替换。

❿ 固定字符数。

下面是同样的例子，但用新方法实现。注意某些位置输出的细微差别：

```
In [77]: 'this is an integer {:d}'.format(15)
Out[77]: 'this is an integer 15'

In [78]: 'this is an integer {:4d}'.format(15)
Out[78]: 'this is an integer 15'

In [79]: 'this is an integer {:04d}'.format(15)
Out[79]: 'this is an integer 0015'

In [80]: 'this is a float {:f}'.format(15.3456)
Out[80]: 'this is a float 15.345600'

In [81]: 'this is a float {:.2f}'.format(15.3456)
```

```
Out[81]: 'this is a float 15.35'

In [82]: 'this is a float {:8f}'.format(15.3456)
Out[82]: 'this is a float 15.345600'

In [83]: 'this is a float {:8.2f}'.format(15.3456)
Out[83]: 'this is a float 15.35'

In [84]: 'this is a float {:08.2f}'.format(15.3456)
Out[84]: 'this is a float 00015.35'

In [85]: 'this is a string {:s}'.format('Python')
Out[85]: 'this is a string Python'

In [86]: 'this is a string {:10s}'.format('Python')
Out[86]: 'this is a string Python '
```

字符串替换在多次打印更新的数据时特别有用，例如在 `while` 循环中：

```
In [87]: i = 0
         while i < 4:
             print('the number is %d' % i)
             i += 1
         the number is 0
         the number is 1
         the number is 2
         the number is 3
In [88]: i = 0
         while i < 4:
             print('the number is {:d}'.format(i))
             i += 1
         the number is 0
         the number is 1
         the number is 2
         the number is 3
```

3.1.6 题外话：正则表达式

正则表达式是处理字符串对象的一个强大工具。Python 在 `re` 模块中提供了这个功能：

```
In [89]: import re
```

假定你现在面对一个大的文本文件，例如逗号分隔值（CSV）文件，其中包含了某些事件序列和相应的日期-时间信息。日期-时间信息多半以 Python 无法直接解释的格式提供。然而，日期-时间信息通常可以通过正则表达式描述。考虑如下的字符串对象，它包含 3

个日期-时间元素、3 个整数和 3 个字符串。注意，3 个引号可以定义多行字符串：

```
In [90]: series = """
         '01/18/2014 13:00:00', 100, '1st';
         '01/18/2014 13:30:00', 110, '2nd';
         '01/18/2014 14:00:00', 120, '3rd'
         """
```

下面的正则表达式描述了上述字符串对象提供的日期-时间信息格式[1]：

```
In [91]: dt = re.compile("'[0-9/:\s]+'") # datetime
```

使用上述正则表达式，我们可以查找所有日期-时间元素。一般来说，对字符串对象应用正则表达式还能带来典型解析任务性能的提升。

```
In [92]: result = dt.findall(series)
         result
Out[92]: ["'01/18/2014 13:00:00'", "'01/18/2014 13:30:00'", "'01/18/2014
         14:00:00'"]
```

正则表达式

解析字符串对象时，若考虑使用正则表达式，可以为相关操作带来便利和高性能。

然后，可以对结果字符串对象进行解析，生成 Python 的日期时间（datetime）对象（用 Python 处理日期和时间数据的概述参见附录 A）。要解析包含日期时间信息的字符串对象，我们必须提供解析方法的信息——同样是一个字符串对象：

```
In [93]: from datetime import datetime
         pydt = datetime.strptime(result[0].replace("'", ""),
                                  '%m/%d/%Y %H:%M:%S')
         pydt
Out[93]: datetime.datetime(2014, 1, 18, 13, 0)

In [94]: print(pydt)
         2014-01-18 13:00:00

In [95]: print(type(pydt))
         <class 'datetime.datetime'>
```

后续内容会提供关于日期-时间数据、这些数据的处理以及 datetime 对象及其方法的更多信息。现在我们只是预览一下金融中的这个重要主题。

1 本书不准备深入细节，但是在互联网上能够找到很多正则表达式以及它们在 Python 中特殊应用的相关信息。这个主题的简介可以参见 Fitzgerald，Michael（2012）：*Introducing Regular Expressions*，O'Reilly，Sebastopol，CA。——原注

3.2 基本数据结构

数据结构是包含其他对象（可能很多）的对象。Python 提供了以下内建结构。

元组（tuple）

　　任意对象的集合；只有少数可用方法。

列表（list）

　　任意对象的集合；有许多可用方法。

字典（dict）

　　键-值存储对象。

集合（set）

　　其他独特对象的无序集合对象。

3.2.1 元组

元组（tuple）是一种高级的数据结构，其应用相当简单有限。它通过圆括号来提供对象定义：

```
In [96]: t = (1, 2.5, 'data')
         type(t)
Out[96]: tuple
```

也可以去掉括号，提供以逗号分隔的多个对象：

```
In [97]: t = 1, 2.5, 'data'
         type(t)
Out[97]: tuple
```

和 Python 中几乎所有的数据结构相似，元组有内建的索引，利用该索引可以读取元组的单个或者多个元素。重要的是记住 Python 使用零起点编号，元组的第 3 个元素在索引位置 2 上：

```
In [98]: t[2]
Out[98]: 'data'

In [99]: type(t[2])
Out[99]: str
```

零起点编号

与 MATLAB 等其他编程语言不同，Python 使用零起点编号方案。例如，元组对象的第一个元素索引值为 0。

这种对象类型只有两个特殊方法：count()和 index()。第一个方法计算某个对象的出现次数，第二个方法给出对象第一次出现位置的索引值：

```
In [100]: t.count('data')
Out[100]: 1

In [101]: t.index(1)
Out[101]: 0
```

元组对象是不可变对象，这意味着一旦定义，它们就不容易更改。

3.2.2　列表

与元组对象相比，列表（list）类型的对象更灵活、更强大。从金融角度看，许多工作只能用列表对象完成，例如存储股票报价和附加新数据。列表对象通过方括号定义，其基本功能和行为与元组类似：

```
In [102]: l = [1, 2.5, 'data']
          l[2]
Out[102]: 'data'
```

列表对象也可以使用 list()函数定义或者转换。下面的代码通过转换前一个例子中的元组对象来生成新的列表对象：

```
In [103]: l = list(t)
          l
Out[103]: [1, 2.5, 'data']

In [104]: type(l)
Out[104]: list
```

除了元组对象的特性之外，列表对象还可以通过不同的方法来扩展和缩小。换句话说，字符串和元组对象是不可变序列对象（具有索引），一经创建不能更改，而列表对象是可变的，可以通过不同操作更改。你可以在现有列表对象中附加一个或者多个列表对象：

```
In [105]: l.append([4, 3]) ❶
          l
Out[105]: [1, 2.5, 'data', [4, 3]]

In [106]: l.extend([1.0, 1.5, 2.0]) ❷
          l
Out[106]: [1, 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]

In [107]: l.insert(1, 'insert') ❸
          l
Out[107]: [1, 'insert', 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]
```

```
In [108]: l.remove('data') ❹
          l
Out[108]: [1, 'insert', 2.5, [4, 3], 1.0, 1.5, 2.0]

In [109]: p = l.pop(3) ❺
          print(l, p)
          [1, 'insert', 2.5, 1.0, 1.5, 2.0] [4, 3]
```

❶ 在尾部附加列表对象。

❷ 附加列表对象元素。

❸ 在索引位置之前插入对象。

❹ 删除第一次出现的对象。

❺ 删除并返回索引位置的对象。

切片操作也很容易实现。这里的切片（slicing）指的是将数据集分解为（感兴趣的）较小部分：

```
In [110]: l[2:5] ❶
Out[110]: [2.5, 1.0, 1.5]
```

❶ 返回的是第 3～5 个元素。

表 3-2 提供了列表对象精选操作和方法的简单说明。

表 3-2　精选的列表对象操作和方法

方法	参数	返回/结果
`l[i] = x`	`[i]`	用 x 替代第 i 个元素
`l[i:j:k] = s`	`[i:j:k]`	用 s 代替从 i 到 j-1 号元素中的每第 k 个元素
`append`	`(x)`	在对象后附加 x
`count`	`(x)`	对象 x 的出现次数
`del l[i:j:k]`	`[i:j:k]`	删除从 i 到 j-1 的索引值的元素
`extend`	`(s)`	将 s 的所有元素附加到对象
`index`	`(x[, i[, j]])`	元素 i 和 j-1 之间第一个 x 的索引
`insert`	`(i, x)++`	在索引 i 之前插入 x
`remove`	`(i)`	删除索引为 i 的元素
`pop`	`(i)`	删除索引为 i 的元素并返回该元素
`reverse`	`()`	颠倒所有项目的顺序
`sort`	`([cmp[, key[, reverse]]])`	对所有项目进行排序

3.2.3 题外话：控制结构

控制结构（如 for 循环）本身就是一个主题，但是在 Python 中，最好在列表对象基础上理解它。这是因为：循环一般来说都在列表对象上进行，这与其他语言中的常用标准有很大不同。看下面的例子，`for` 循环在列表对象 l 索引值为 2～4 的元素上进行，功能是打印各个元素的平方。注意第二行中缩进（空白）的重要性：

```
In [111]: for element in l[2:5]:
              print(element ** 2)
          6.25
          1.0
          2.25
```

与典型的基于计数器的循环相比，上述循环提供了很高的灵活性。基于计数器的循环也是 Python 的一个可选项，但是在（标准）列表对象 `range` 的基础上完成的：

```
In [112]: r = range(0, 8, 1) ❶
          r
Out[112]: range(0, 8)

In [113]: type(r)
Out[113]: range
```

❶ 参数依次是起始值、结束值和步长。

为了比较，下面我们使用 `range` 实现同一个循环：

```
In [114]: for i in range(2, 5):
              print(l[i] ** 2)
          6.25
          1.0
          2.25
```

在列表上循环

在 Python 中，你可以在任意列表对象上循环，而不用管这些对象的内容是什么，这往往可以避免引入计数器。

Python 还提供了典型（条件）控制元素 `if`、`elif` 和 `else`。它们的用途与其他语言的相同：

```
In [115]: for i in range(1, 10):
              if i % 2 == 0: ❶
                  print("%d is even" % i)
              elif i % 3 == 0:
                  print("%d is multiple of 3" % i)
```

```
        else:
            print("%d is odd" % i)
1 is odd
2 is even
3 is multiple of 3
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is multiple of 3
```

❶ %代表取模运算。

同样，`while` 提供了另一种流程控制手段：

```
In [116]: total = 0
          while total < 100:
              total += 1
          print(total)
          100
```

Python 的特殊性之一是所谓的列表推导（List Comprehension）。这种方法不在现有列表对象上循环，而是以紧凑的方式通过循环生成列表对象：

```
In [117]: m = [i ** 2 for i in range(5)]
          m
Out[117]: [0, 1, 4, 9, 16]
```

在某种意义上，它已经提供了一种方法，以较为隐含（而非显式）的循环生成某种向量化代码段（代码向量化将在第 4 章和第 5 章中详细讨论）。

3.2.4 题外话：函数式编程

Python 也提供一些用于函数式编程支持的工具——在一整组输入（在我们的例子中是列表对象）上应用某个函数。这些工具是过滤（filter）、映射（map）和归纳（reduce）。但是，我们首先需要一个函数定义。从简单的功能出发，考虑以下的函数 `f`，它返回输入 `x` 的平方数：

```
In [118]: def f(x):
              return x ** 2
          f(2)
Out[118]: 4
```

当然，函数可能很复杂，它可能有多个输入（参数对象）甚至多个输出（返回对象）。考虑以下函数：

```
In [119]: def even(x):
              return x % 2 == 0
          even(3)
Out[119]: False
```

返回对象是一个布尔值。这样的函数可以用 `map()` 应用到整个列表对象：

```
In [120]: list(map(even, range(10)))
Out[120]: [True, False, True, False, True, False, True, False, True, False]
```

为此，我们还可以使用 `lambda` 或者匿名函数，直接提供一个函数定义作为 `map()` 的参数：

```
In [121]: list(map(lambda x: x ** 2, range(10)))
Out[121]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

函数也可以用于过滤列表对象。在下面的例子中，过滤器返回列表对象中匹配某个布尔条件（由 `even` 函数定义）的元素：

```
In [122]: list(filter(even, range(15)))
Out[122]: [0, 2, 4, 6, 8, 10, 12, 14]
```

列表推导、函数式编程和匿名函数

在 Python 级别上尽可能避免循环被视为“好的习惯”。列表推导和函数式编程工具（如 `map`、`filter` 和 `reduce`）为编写紧凑（一般来说也更易于理解）无循环代码提供了很大帮助。`lambda` 或者匿名函数也是这方面的强大工具。

3.2.5 字典

字典（dict）对象就是可以按照键码（例如，可能是字符串对象）读取的数据字典，也是一种可变序列，是所谓的键—值存储。列表对象是有序且可排序的，而字典对象通常是无序、不可排序的[1]。示例最能够说明字典和列表对象的不同之处。字典对象定义用花括号实现：

```
In [123]: d = {
               'Name' : 'Angela Merkel',
               'Country' : 'Germany',
               'Profession' : 'Chancelor',
               'Age' : 64
               }
          type(d)
Out[123]: dict

In [124]: print(d['Name'], d['Age'])
          Angela Merkel 64
```

1　标准字典有一些变种，其中包括 OederedDict 子类，它可以记忆添加字典项的顺序。

同样，该类对象也有一些内建方法：

```
In [125]: d.keys()
Out[125]: dict_keys(['Name', 'Country', 'Profession', 'Age'])

In [126]: d.values()
Out[126]: dict_values(['Angela Merkel', 'Germany', 'Chancelor', 64])

In [127]: d.items()
Out[127]: dict_items([('Name', 'Angela Merkel'), ('Country', 'Germany'),
          ('Profession', 'Chancelor'), ('Age', 64)])

In [128]: birthday = True
          if birthday:
              d['Age'] += 1
          print(d['Age'])
          65
```

从字典对象中获得迭代器（Iterator）对象有多种方法。这种对象在循环时的表现与列表对象的表现类似：

```
In [129]: for item in d.items():
              print(item)
          ('Name', 'Angela Merkel')
          ('Country', 'Germany')
          ('Profession', 'Chancelor')
          ('Age', 65)

In [130]: for value in d.values():
              print(type(value))
          <class 'str'>
          <class 'str'>
          <class 'str'>
          <class 'int'>
```

表 3-3 提供了字典对象上精选的操作和方法的简单说明。

表 3-3　字典对象精选操作和方法

方法	参数	返回/结果
d[k]	[k]	d 中键码为 k 的项目
d[k] = x	[k]	将键码为 k 的项目设置为 x
del d[k]	[k]	删除键码为 k 的项目
clear	()	删除所有项目

续表

方法	参数	返回/结果
copy	()	建立一个副本
has_key	(k)	如果 k 是一个键码，为真
items	()	所有键-值对的副本
keys	()	所有键码的迭代器
values	()	所有值的迭代器
poptiem	(k)	返回并删除键码为 k 的项目
update	([e])	用来自 e 的项目更新字典项目

3.2.6 集合

本节介绍的最后一种数据结构是集合（set）对象。虽然集合论是数学和金融理论的基石，但是集合对象并没有太多实际应用。这种对象是其他对象的无序集合，每个元素只包含一次：

```
In [131]: s = set(['u', 'd', 'ud', 'du', 'd', 'du'])
          s
Out[131]: {'d', 'du', 'u', 'ud'}

In [132]: t = set(['d', 'dd', 'uu', 'u'])
```

使用集合对象可以实现数学集合论中进行的运算。例如，可以生成并集、交集和差集：

```
In [133]: s.union(t) ❶
Out[133]: {'d', 'dd', 'du', 'u', 'ud', 'uu'}

In [134]: s.intersection(t) ❷
Out[134]: {'d', 'u'}

In [135]: s.difference(t) ❸
Out[135]: {'du', 'ud'}

In [136]: t.difference(s) ❹
Out[136]: {'dd', 'uu'}

In [137]: s.symmetric_difference(t) ❺
Out[137]: {'dd', 'du', 'ud', 'uu'}
```

❶ s 和 t 的所有元素。

❷ s 和 t 中的所有项目。

❸ 在 s 中而不在 t 中的项目。

❹ 在 t 中但不在 s 中的项目。

❺ 在 s 或者 t 中、但不同时在两个集合中的项目。

集合对象的应用之一是去掉列表对象中的重复数据，例如：

```
In [138]: from random import randint
          l = [randint(0, 10) for i in range(1000)] ❶
          len(l) ❷
Out[138]: 1000

In [139]: l[:20]
Out[139]: [4, 2, 10, 2, 1, 10, 0, 6, 0, 8, 10, 9, 2, 4, 7, 8, 10, 8, 8, 2]

In [140]: s = set(l)
          s
Out[140]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

❶ 0～10 之间的 1000 个随机整数。

❷ l 中的元素数量。

3.3 结语

Python 基本解释程序提供了一组丰富灵活的数据结构。从金融的角度看，下面这些类型是比较重要的。

基本数据类型

在金融中，整数、浮点数、布尔值和字符串提供了原子数据类型。

标准数据结构

元组、列表、字典和集合类在金融中有许多应用领域，列表通常是数值用例中灵活的多面手。

3.4 延伸阅读

本章聚焦于对金融算法和应用特别重要的数据类型和结构。然而，这只是 Python 中数据结构和数据建模探索的出发点。

有许多宝贵的资源深入地探讨了这些问题，详情可参见 Python 数据结构的官方文档。

下面是一些好的参考书。

- Goodrich, Michael, et al. (2013). *Data Structures and Algorithms in Python.* Hoboken, NJ: John Wiley & Sons.
- Harrison, Matt (2017). *Illustrated Guide to Python 3.* CreateSpace Treading on Python Series.
- Ramalho, Luciano (2016). *Fluent Python*. Sebastopol, CA: O'Reilly.

正则表达式的介绍请参见。

- Fitzgerald, Michael (2012). *Introducing Regular Expressions*. Sebastopol, CA: O'Reilly.

第 4 章

用 NumPy 进行数值计算

计算机是没有用的，它们只能给出答案。

——巴勃罗·毕加索

尽管 Python 解释程序已经带来了非常丰富的数据结构，但 NumPy 和其他库自有的数据结构也非常有价值。本章主要介绍 NumPy，这个库提供了保存同构与异构数据集合的多维数组对象，并支持代码向量化。

本章介绍以下数据结构。

对象类型	含义	用途
`ndrray`（常规）	*n* 维数组对象	大规模数值数据
`ndrray`（记录）	二维数组	组织成列的表格数据

本章的组织如下。

数据数组

本节介绍用纯 Python 代码处理数据数组的方法。

常规 NumPy 数组

这是关于常规 `NumPy ndarray` 类的核心部分，在几乎所有涉及数值数据的数据密集型 Python 用例中，该类都担纲主力。

NumPy 结构数组

这一节介绍结构化（或者记录）`ndarray` 对象，该对象用于处理分列表格数据。

代码向量化

这一节讨论代码向量化及其好处，还将讨论某些场景下内存布局的重要性。

4.1 数据数组

第 3 章介绍了 Python 提供的一些相当实用而灵活的通用数据结构，特别是列表对象，它可以被视为真正的主力干将，具有许多方便的特性，可应用于多个应用领域。然而，科学和金融应用通常有在特殊数据结构上进行高性能操作的需求。这方面最重要的数据结构之一是数组。数组通常在行和列中存放相同数据类型的其他（基本）对象。

现在假定我们只使用数值，但是这些概念可以推广到其他类型的数据。在最简单的情况下，一维数组从数学上讲，一般可以表示一个实数向量，这些实数的内部表示是浮点对象。一维数组只表示单行或者单列元素。在更常见的情况下，数组可以表示一个 $i \times j$ 个元素的矩阵。这一概念可以推广到 $i \times j \times k$ 个元素的立方矩阵（3 维数组），以及包含 $i \times j \times k \times l \times \cdots$个元素的 n 维数组。

线性代数和向量空间理论等数学课程阐述了这种数学结构在许多学科和领域中的重要性，也就证明了专门为方便、高效地处理数组设计而提供的数据结构是卓有成效的。这些领域也是 Python 的 NumPy 库及其 ndarray 类大展身手的领域。在介绍这个类之前，先介绍处理数组的另外两种替代方法。

4.1.1 用 Python 列表形成数组

数组可以用第 3 章介绍的内建数据结构构造。列表对象特别适合于完成这项任务。简单的列表已经可以视为一维数组：

```
In [1]: v = [0.5, 0.75, 1.0, 1.5, 2.0] ❶
```

❶ 包含数值的列表对象。

由于列表对象可能包含任意的其他对象，它们也可以包含其他列表对象。这样，二维数组和更高维的数组很容易通过嵌套列表对象来构造：

```
In [2]: m = [v, v, v] ❶
        m ❷
Out[2]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]
```

❶ 包含列表对象的列表对象。

❷ 构成一个数值矩阵。

我们也可以很轻松地通过简单的索引选择行，通过双重索引访问单个元素（但是，选择整列不容易）：

```
In [3]: m[1]
Out[3]: [0.5, 0.75, 1.0, 1.5, 2.0]
```

```
In [4]: m[1][0]
Out[4]: 0.5
```

嵌套可以更进一步，以获得更通用的结构：

```
In [5]: v1 = [0.5, 1.5]
        v2 = [1, 2]
        m = [v1, v2]
        c = [m, m] ❶
        c
Out[5]: [[[0.5, 1.5], [1, 2]], [[0.5, 1.5], [1, 2]]]

In [6]: c[1][1][0]
Out[6]: 1
```

❶ 多维数组。

注意，按照刚才介绍的方式所组合的对象通常使用原始对象的引用指针。这在实践中有何意义？我们来看看以下的操作：

```
In [7]: v = [0.5, 0.75, 1.0, 1.5, 2.0]
        m = [v, v, v]
        m
Out[7]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]
```

现在，改变 v 对象的第一个元素，看看 m 对象发生了什么：

```
In [8]: v[0] = 'Python'
        m
Out[8]: [['Python', 0.75, 1.0, 1.5, 2.0],
         ['Python', 0.75, 1.0, 1.5, 2.0],
         ['Python', 0.75, 1.0, 1.5, 2.0]]
```

使用 copy 模块的 deepcopy 函数可以避免这一现象：

```
In [9]: from copy import deepcopy
        v = [0.5, 0.75, 1.0, 1.5, 2.0]
        m = 3 * [deepcopy(v), ] ❶
        m
Out[9]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]

In [10]: v[0] = 'Python' ❷
```

```
         m ❸
Out[10]: [[0.5, 0.75, 1.0, 1.5, 2.0],
          [0.5, 0.75, 1.0, 1.5, 2.0],
          [0.5, 0.75, 1.0, 1.5, 2.0]]
```

❶ 用物理副本代替引用指针。

❷ 结果是原始对象的改变。

❸ 不再对 m 有任何影响。

4.1.2　Python array 类

Python 中有一个专门的 array 模块，下面是 Python 官方文档中的说明。

> 该模块定义一个对象类型，对象类型以紧凑的方式表示基本值的数组：字符、整数、浮点数。数组是顺序类型，其表现与列表非常相似，但保存在数组中的对象类型受到限制。类型在对象创建时用一个单字符的类型代码来指定。

考虑如下代码，它通过一个列表对象来实例化数组（array）对象：

```
In [11]: v = [0.5, 0.75, 1.0, 1.5, 2.0]

In [12]: import array

In [13]: a = array.array('f', v) ❶
         a
Out[13]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0])

In [14]: a.append(0.5) ❷
         a
Out[14]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5])

In [15]: a.extend([5.0, 6.75]) ❷
         a
Out[15]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75])
In [16]: 2 * a ❸
Out[16]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75, 0.5,
         0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75])
```

❶ 以类型代码 f（表示浮点数）实例化数组对象。

❷ 主要方法与列表对象类似。

❸ 虽然“标量乘法”理论上可行，但结果与数学理论不符，它只是重复元素。

附加与指定数据类型不同的对象将导致 TypeError 错误：

```
In [17]: a.append('string') ❶

         ---------------------------------------
         TypeErrorTraceback (most recent call last)
         <ipython-input-17-14cd6281866b> in <module>()
         ----> 1 a.append('string') ❶

         TypeError: must be real number, not str

In [18]: a.tolist() ❷
Out[18]: [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75]
```

❶ 只能附加浮点数对象，其他数据类型或类型代码将导致出错。

❷ 不过，如果有必要，数组对象很容易轻松地转换回列表对象。

array 类的优势之一是具有内建存储及检索功能：

```
In [19]: f = open('array.apy', 'wb') ❶
         a.tofile(f) ❷
         f.close() ❸

In [20]: with open('array.apy', 'wb') as f: ❹
             a.tofile(f) ❹

In [21]: !ls -n arr* ❺
         -rw-r--r--@ 1 503 20 32 Nov 7 11:46 array.apy
```

❶ 打开一个磁盘文件来写入二进制数据。

❷ 将数组数据写到文件中。

❸ 关闭文件。

❹ 替代方法：用 with 上下文完成相同操作。

❺ 显示写入磁盘的文件。

和前面一样，从磁盘读取数据时，array 对象的数据类型很重要：

```
In [22]: b = array.array('f') ❶

In [23]: with open('array.apy', 'rb') as f: ❷
             b.fromfile(f, 5) ❸

In [24]: b ❹
Out[24]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0])

In [25]: b = array.array('d') ❹
```

```
In [26]: with open('array.apy', 'rb') as f:
             b.fromfile(f, 2) ❺

In [27]: b ❻
Out[27]: array('d', [0.0004882813645963324, 0.12500002956949174])
```

❶ 用类型码 `float` 初始化新的数组对象。

❷ 打开文件来读取二进制数据。

❸ 将 5 个元素读入 `b` 对象。

❹ 用类型码 `double` 初始化一个新的数组对象。

❺ 从文件中读取两个元素。

❻ 类型码的不同导致"错误"的数字。

4.2 常规 NumPy 数组

用列表对象组成数组结构是可行的。但是这实现起来并不方便，`list` 类本身不是为这一特殊目标设计的，它是为了更广泛、更通用的目的而构建的。`array` 类更专业一些，提供了处理数据数组的一些实用功能。但是专用的类能够真正地为处理数组类型结构带来好处。

4.2.1 基础知识

`numpy.ndarray` 就是专用类，它能够方便、高效（也就是高性能）地处理 *n* 维数组。同样，最好是用示例来说明这种类的使用方式：

```
In [28]: import numpy as np ❶

In [29]: a = np.array([0, 0.5, 1.0, 1.5, 2.0]) ❷
         a
Out[29]: array([0. , 0.5, 1. , 1.5, 2. ])

In [30]: type(a) ❷
Out[30]: numpy.ndarray

In [31]: a = np.array(['a', 'b', 'c']) ❸
         a
Out[31]: array(['a', 'b', 'c'], dtype='<U1')

In [32]: a = np.arange(2, 20, 2) ❹
```

```
         a
Out[32]: array([ 2, 4, 6, 8, 10, 12, 14, 16, 18])

In [33]: a = np.arange(8, dtype=np.float) ❺
         a
Out[33]: array([0., 1., 2., 3., 4., 5., 6., 7.])

In [34]: a[5:] ❻
Out[34]: array([5., 6., 7.])

In [35]: a[:2] ❻
Out[35]: array([0., 1.])
```

❶ 导入 numpy 软件包。

❷ 从包含浮点数的列表对象中创建一个 ndarray 对象。

❸ 从包含字符串的列表对象中创建一个 ndarray 对象。

❹ np.arange()与 range()的工作方式类似。

❺ np.arange()增加了一个输入参数 dtype。

❻ 对于一维 ndarray 对象，索引方式不变。

numpy.ndarray 类的主要特征之一是它有多种内建方法，例如：

```
In [36]: a.sum() ❶
Out[36]: 28.0

In [37]: a.std() ❷
Out[37]: 2.29128784747792

In [38]: a.cumsum() ❸
Out[38]: array([ 0., 1., 3., 6., 10., 15., 21., 28.])
```

❶ 所有元素的总和。

❷ 元素的标准差。

❸ 所有元素的累计总和（从索引位置 0 起）。

另一个重要特征是在 ndarray 对象上定义的（向量化）数学运算：

```
In [39]: l = [0., 0.5, 1.5, 3., 5.]
         2 * l ❶
Out[39]: [0.0, 0.5, 1.5, 3.0, 5.0, 0.0, 0.5, 1.5, 3.0, 5.0]

In [40]: a
Out[40]: array([0., 1., 2., 3., 4., 5., 6., 7.])
```

```
In [41]: 2 * a ❷
Out[41]: array([ 0., 2., 4., 6., 8., 10., 12., 14.])

In [42]: a ** 2 ❸
Out[42]: array([ 0., 1., 4., 9., 16., 25., 36., 49.])

In [43]: 2 ** a ❹
Out[43]: array([ 1., 2., 4., 8., 16., 32., 64., 128.])

In [44]: a ** a ❺
Out[44]: array([1.00000e+00, 1.00000e+00, 4.00000e+00, 2.70000e+01,
                2.56000e+02, 3.12500e+03, 4.66560e+04, 8.23543e+05])
```

❶ 对列表对象使用标量乘法来复制元素。

❷ 相反，ndarray 对象可以实现正确的标量乘法。

❸ 计算每个元素的平方值。

❹ 将 ndarray 元素作为指数求取幂值。

❺ 求取将每个元素作为底和指数的幂值。

通用函数是 NumPy 软件包的另一个重要特性。“通用”指的是，它们一般既可以用在 ndarray 对象上，也可用于 Python 的基本数据类型上。但当通用函数应用到某个 Python 对象（如 float）时，使用者必须意识到，其性能不如 math 模块中的可实现相同目标的功能：

```
In [45]: np.exp(a) ❶
Out[45]: array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00,
              2.00855369e+01, 5.45981500e+01, 1.48413159e+02,
              4.03428793e+02, 1.09663316e+03])

In [46]: np.sqrt(a) ❷
Out[46]: array([0.        , 1. , 1.41421356, 1.73205081, 2. ,
                2.23606798, 2.44948974, 2.64575131])

In [47]: np.sqrt(2.5) ❸
Out[47]: 1.5811388300841898

In [48]: import math ❹

In [49]: math.sqrt(2.5) ❹
Out[49]: 1.5811388300841898

In [50]: math.sqrt(a) ❺
```

```
-------------------------------------
TypeErrorTraceback (most recent call last)
<ipython-input-50-b39de4150838> in <module>()
----> 1 math.sqrt(a) ❺

TypeError: only size-1 arrays can be converted to Python scalars

In [51]: %timeit np.sqrt(2.5) ❻
         722 ns ± 13.7 ns per loop (mean ± std. dev. of 7 runs,
         1000000 loops each)

In [52]: %timeit math.sqrt(2.5) ❼
         91.8 ns ± 4.13 ns per loop (mean ± std. dev. of 7 runs,
         10000000 loops each)
```

❶ 计算每个元素的指数值。

❷ 计算每个元素的平方根。

❸ 计算 Python 浮点对象的平方根。

❹ 使用 `math` 模块进行相同的计算。

❺ `math.sqrt()`函数不能直接应用到 `ndarray` 对象。

❻ 将通用函数 `np.sqrt()`应用到 Python 浮点对象。

❼ 其速度远慢于用 `math.sqrt()`函数进行的相同运算。

4.2.2 多维数组

一维以上的数组可以无缝过渡，迄今为止介绍的功能都可以用于更一般的示例，特别是当所有的维度都有一致的索引方式时：

```
In [53]: b = np.array([a, a * 2]) ❶
         b
Out[53]: array([[ 0., 1., 2., 3., 4., 5., 6., 7.],
                [ 0., 2., 4., 6., 8., 10., 12., 14.]])

In [54]: b[0] ❷
Out[54]: array([0., 1., 2., 3., 4., 5., 6., 7.])

In [55]: b[0, 2] ❸
Out[55]: 2.0

In [56]: b[:, 1] ❹
Out[56]: array([1., 2.])
```

```
In [57]: b.sum() ❺
Out[57]: 84.0

In [58]: b.sum(axis=0) ❻
Out[58]: array([ 0., 3., 6., 9., 12., 15., 18., 21.])

In [59]: b.sum(axis=1) ❼
Out[59]: array([28., 56.])
```

❶ 从一维 ndarray 对象构造二维对象。

❷ 选择第 1 行。

❸ 选择第 1 行的第 3 个元素；索引在方括号中，以逗号分隔。

❹ 选择第 2 列。

❺ 计算所有值的总和。

❻ 沿第 1 个轴（即按列）计算总值。

❼ 沿第 2 个轴（按行）计算总值。

初始化（实例化）ndarray 对象有许多种方式。前面已经介绍了一种——通过 np.array。但是，这种方法有个前提，即所有数组元素已经可用。但是，有人可能喜欢先实例化 ndarray 对象，然后再用代码执行期间生成的结果填充数组。为此，我们可以使用如下代码：

```
In [60]: c = np.zeros((2, 3), dtype='i', order='C') ❶
         c
Out[60]: array([[0, 0, 0],
                [0, 0, 0]], dtype=int32)

In [61]: c = np.ones((2, 3, 4), dtype='i', order='C') ❷
         c
Out[61]: array([[[1, 1, 1, 1],
                 [1, 1, 1, 1],
                 [1, 1, 1, 1]],

                [[1, 1, 1, 1],
                 [1, 1, 1, 1],
                 [1, 1, 1, 1]]], dtype=int32)

In [62]: d = np.zeros_like(c, dtype='f16', order='C') ❸
         d
Out[62]: array([[[0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]],
```

```
                [[0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]]], dtype=float128)

In [63]: d = np.ones_like(c, dtype='f16', order='C') ❸
         d
Out[63]: array([[[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]],

                [[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]]], dtype=float128)

In [64]: e = np.empty((2, 3, 2)) ❹
         e
Out[64]: array([[[0.00000000e+000, 0.00000000e+000],
                 [0.00000000e+000, 0.00000000e+000],
                 [0.00000000e+000, 0.00000000e+000]],

                [[0.00000000e+000, 0.00000000e+000],
                 [0.00000000e+000, 7.49874326e+247],
                 [1.28822975e-231, 4.33190018e-311]]])
In [65]: f = np.empty_like(c) ❹
         f
Out[65]: array([[[         0,          0,          0,          0],
                 [         0,          0,          0,          0],
                 [         0,          0,          0,          0]],

                [[         0,          0,          0,          0],
                 [         0,          0,  740455269, 1936028450],
                 [         0,  268435456, 1835316017,       2041]]],
                dtype=int32)
In [66]: np.eye(5) ❺
Out[66]: array([[1., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0.],
                [0., 0., 0., 1., 0.],
                [0., 0., 0., 0., 1.]])

In [67]: g = np.linspace(5, 15, 12) ❻
         g
Out[67]: array([ 5.        ,  5.90909091,  6.81818182,  7.72727273,  8.63636364,
                 9.54545455, 10.45454545, 11.36363636, 12.27272727,
                13.18181818, 14.09090909, 15.        ])
```

❶ 创建一个 `ndarray` 对象，预先为其填充 0。

❷ 创建一个 `ndarray` 对象，预先为其填充 1。

❸ 同上，但通过另一个 `ndarray` 对象来推导数组的构成。

❹ 创建一个不预先填充任何数值的 `ndarray` 对象（数值取决于内存中的对应位）。

❺ 创建一个代表方阵的 `ndarray` 对象，在对角线位置填充 1。

❻ 创建一维 `ndarray` 对象，数值之间有均匀的间隔；参数为 `start`（起始值）、`end`（结束值）和 `num`（元素数量）。

这些函数提供了如下信息。

组成

整数（int）、整数序列或者引用另一个 `ndarray`。

dtype（可选）

dtype——用于 `ndarray` 对象的 NumPy 特定数据类型。

order（顺序，可选）

在内存中存储元素的顺序：C 指和 C 语言相似（行优先），F 指和 Fortran 类似（列优先）。

现在，和基于列表的方法相比，NumPy 用 `ndarray` 类构造数组的方式的特殊之处已经很明显：

- `ndarray` 对象有内建的维度（轴）；
- `ndarray` 对象是不可变的，其长度（大小）固定；
- 整个数组只允许一种数据类型（`np.dtype`）。

相比之下，`class` 类的共同特性只有一个：只允许单一数据类型（类型码为 `dtype`）。

`order` 参数的作用将在本章后面讨论。表 4-1 提供了 `np.dtype` 对象的简介（即 NumPy 可以使用的基本数据类型）。

表 4-1　NumPy dtype 对象

dtype	描述	示例
?	布尔值	?（`true` 或者 `false`）
`i`	有符号整数	`i8`（64 位）
`u`	无符号整数	`u8`（64 位）
`f`	浮点数	`f8`（64 位）
`c`	浮点复数	`c32`（256 位）

续表

dtype	描述	示例
m	时间差	m（64 位）
M	日期时间	M（64 位）
O	对象	O（指向对象的指针）
U	Unicode	U24（24 个 Unicode 字符）
V	其他	V12（12 字节数据块）

4.2.3　元信息

每个 ndarray 对象都提供了一些有用的属性：

```
In [68]: g.size ❶
Out[68]: 12

In [69]: g.itemsize ❷
Out[69]: 8

In [70]: g.ndim ❸
Out[70]: 1

In [71]: g.shape ❹
Out[71]: (12,)

In [72]: g.dtype ❺
Out[72]: dtype('float64')

In [73]: g.nbytes ❻
Out[73]: 96
```

❶ 元素数量。

❷ 用于表示一个元素的字节数。

❸ 维数。

❹ ndarray 对象组成。

❺ 元素的 dtype。

❻ 在内存中使用的字节总数。

4.2.4　改变组成与大小

虽然 ndarray 对象默认是不可变的，但可以改变这种对象的组成和大小，方法有多种。

改变组成通常只是提供对同一组数据的另一种视图，改变大小通常是创建一个新（临时）对象。首先介绍一些改变组成的例子：

```
In [74]: g = np.arange(15)

In [75]: g
Out[75]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])

In [76]: g.shape ❶
Out[76]: (15,)

In [77]: np.shape(g) ❶
Out[77]: (15,)

In [78]: g.reshape((3, 5)) ❷
Out[78]: array([[ 0, 1, 2, 3, 4],
                [ 5, 6, 7, 8, 9],
                [10, 11, 12, 13, 14]])

In [79]: h = g.reshape((5, 3)) ❸
         h
Out[79]: array([[ 0, 1, 2],
                [ 3, 4, 5],
                [ 6, 7, 8],
                [ 9, 10, 11],
                [12, 13, 14]])

In [80]: h.T ❹
Out[80]: array([[ 0, 3, 6, 9, 12],
                [ 1, 4, 7, 10, 13],
                [ 2, 5, 8, 11, 14]])

In [81]: h.transpose() ❹
Out[81]: array([[ 0, 3, 6, 9, 12],
                [ 1, 4, 7, 10, 13],
                [ 2, 5, 8, 11, 14]])
```

❶ 原始 `ndarray` 对象的组成。

❷ 将对象组成改为二维（内存视图）。

❸ 创建一个新对象。

❹ 新 `ndarray` 对象的转置。

在改变组成的操作中，`ndarray` 对象中的元素总数不变。在改变大小的操作中，这一数量改变——可能减少（缩小）或者增加（扩大）。下面是改变大小的一些例子：

```
In [82]: g
Out[82]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])

In [83]: np.resize(g, (3, 1)) ❶
Out[83]: array([[0],
                [1],
                [2]])

In [84]: np.resize(g, (1, 5)) ❶
Out[84]: array([[0, 1, 2, 3, 4]])

In [85]: np.resize(g, (2, 5)) ❶
Out[85]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])

In [86]: n = np.resize(g, (5, 4)) ❷
         n
Out[86]: array([[ 0, 1, 2, 3],
                [ 4, 5, 6, 7],
                [ 8, 9, 10, 11],
                [12, 13, 14, 0],
                [ 1, 2, 3, 4]])
```

❶ 缩小二维数组。

❷ 扩大二维数组。

堆叠是一种特殊操作，可以将两个 ndarray 对象从水平或者垂直方向组合起来。但是，“连接”的对象的维的大小必须相同：

```
In [87]: h
Out[87]: array([[ 0, 1, 2],
                [ 3, 4, 5],
                [ 6, 7, 8],
                [ 9, 10, 11],
                [12, 13, 14]])

In [88]: np.hstack((h, 2 * h)) ❶
Out[88]: array([[ 0, 1, 2, 0, 2, 4],
                [ 3, 4, 5, 6, 8, 10],
                [ 6, 7, 8, 12, 14, 16],
                [ 9, 10, 11, 18, 20, 22],
                [12, 13, 14, 24, 26, 28]])

In [89]: np.vstack((h, 0.5 * h)) ❷
Out[89]: array([[ 0. , 1. , 2. ],
```

```
                [ 3. , 4. , 5. ],
                [ 6. , 7. , 8. ],
                [ 9. , 10. , 11. ],
                [12. , 13. , 14. ],
                [ 0. , 0.5, 1. ],
                [ 1.5, 2. , 2.5],
                [ 3. , 3.5, 4. ],
                [ 4.5, 5. , 5.5],
                [ 6. , 6.5, 7. ]])
```

❶ 两个 ndarray 对象进行水平堆叠。

❷ 两个 ndarray 对象进行垂直堆叠。

另一种特殊操作是对多维 ndarray 对象进行扁平化操作，使其成为一维对象。我们可以选择按照行（C 顺序）或者列（F 顺序）进行扁平化：

```
In [90]: h
Out[90]: array([[ 0, 1, 2],
                [ 3, 4, 5],
                [ 6, 7, 8],
                [ 9, 10, 11],
                [12, 13, 14]])

In [91]: h.flatten() ❶
Out[91]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])

In [92]: h.flatten(order='C') ❶
Out[92]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])

In [93]: h.flatten(order='F') ❷
Out[93]: array([ 0, 3, 6, 9, 12, 1, 4, 7, 10, 13, 2, 5, 8, 11, 14])

In [94]: for i in h.flat: ❸
             print(i, end=',')
         0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
In [95]: for i in h.ravel(order='C'): ❹
             print(i, end=',')
         0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
In [96]: for i in h.ravel(order='F'): ❹
             print(i, end=',')
         0,3,6,9,12,1,4,7,10,13,2,5,8,11,14,
```

❶ 默认扁平化顺序为 C。

❷ 以 F 顺序进行扁平化。

❸ `flat` 属性提供扁平迭代器（C 顺序）。

❹ `ravel()`是 `flatten()`的替代方法。

4.2.5 布尔数组

`ndarray` 对象上的比较和逻辑运算通常和标准 Python 数据类型一样：按元素顺序进行。对条件进行求值的默认结果是一个布尔型 `ndarray` 对象（`dtype` 为 `bool`）：

```
In [97]: h
Out[97]: array([[ 0, 1, 2],
                [ 3, 4, 5],
                [ 6, 7, 8],
                [ 9, 10, 11],
                [12, 13, 14]])

In [98]: h > 8 ❶
Out[98]: array([[False, False, False],
                [False, False, False],
                [False, False, False],
                [ True, True, True],
                [ True, True, True]])

In [99]: h <= 7 ❷
Out[99]: array([[ True, True, True],
                [ True, True, True],
                [ True, True, False],
                [False, False, False],
                [False, False, False]])

In [100]: h == 5 ❸
Out[100]: array([[False, False, False],
                 [False, False, True],
                 [False, False, False],
                 [False, False, False],
                 [False, False, False]])

In [101]: (h == 5).astype(int) ❹
Out[101]: array([[0, 0, 0],
                 [0, 0, 1],
                 [0, 0, 0],
                 [0, 0, 0],
                 [0, 0, 0]])

In [102]: (h > 4) & (h <= 12) ❺
Out[102]: array([[False, False, False],
```

```
                [False, False, True],
                [ True, True, True],
                [ True, True, True],
                [ True, False, False]])
```

❶ 值大于 A 吗？

❷ 值小于或等于 B 吗？

❸ 值等于 C 吗？

❹ 以整数 0 和 1 代表 `True` 和 `False`。

❺ 值大于 D 且小于或等于 E 吗？

这种布尔数组可用于索引和数据选择。例如，如下对数据的扁平化操作：

```
In [103]: h[h > 8] ❶
Out[103]: array([ 9, 10, 11, 12, 13, 14])

In [104]: h[(h > 4) & (h <= 12)] ❷
Out[104]: array([ 5, 6, 7, 8, 9, 10, 11, 12])

In [105]: h[(h < 4) | (h >= 12)] ❸
Out[105]: array([ 0, 1, 2, 3, 12, 13, 14])
```

❶ 所有大于 A 的值。

❷ 所有大于 B 且小于等于 C 的值。

❸ 所有小于 D 或大于等于 E 的值。

`np.where()`函数是这方面的一个强大工具，可以根据某个条件是 `True` 或者 `False` 来定义操作。应用 `np.where()`生成的结果是一个与原始对象有着相同组成形式的新 `ndarray` 对象：

```
In [106]: np.where(h > 7, 1, 0) ❶
Out[106]: array([[0, 0, 0],
                [0, 0, 0],
                [0, 0, 1],
                [1, 1, 1],
                [1, 1, 1]])

In [107]: np.where(h % 2 == 0, 'even', 'odd') ❷
Out[107]: array([['even', 'odd', 'even'],
                ['odd', 'even', 'odd'],
                ['even', 'odd', 'even'],
                ['odd', 'even', 'odd'],
                ['even', 'odd', 'even']], dtype='<U4')
```

```
In [108]: np.where(h <= 7, h * 2, h / 2) ❸
Out[108]: array([[ 0. , 2. , 4. ],
                 [ 6. , 8. , 10. ],
                 [12. , 14. , 4. ],
                 [ 4.5, 5. , 5.5],
                 [ 6. , 6.5, 7. ]])
```

❶ 在新对象中，如果条件为真则设置为 1，否则设置为 0。

❷ 在新对象中，如果条件为真则设置为 even，否则设置为 odd。

❸ 在新对象中，如果条件为真则设置为 h 元素的两倍，否则设置为 h 元素的一半。

后面几章将提供 ndarray 对象重要操作的更多例子。

4.2.6 速度对比

稍后，我们将介绍 NumPy 结构化数组，但在介绍之前，先来看看这种专业化的常规数组能带来什么样的性能。

做个简单的练习，假定我们要生成一个 5000×5000 个元素组成的矩阵/数组，在矩阵中填入标准正态分布的（伪）随机数，然后计算所有元素的总和。首先采用纯 Python 方法，使用列表推导：

```
In [109]: import random
          I = 5000

In [110]: %time mat = [[random.gauss(0, 1) for j in range(I)] \
                       for i in range(I)] ❶
          CPU times: user 17.1 s, sys: 361 ms, total: 17.4 s
          Wall time: 17.4 s

In [111]: mat[0][:5] ❷
Out[111]: [-0.40594967782329183,
           -1.357757478015285,
           0.05129566894355976,
           -0.8958429976582192,
           0.6234174778878331]
In [112]: %time sum([sum(l) for l in mat]) ❸
          CPU times: user 142 ms, sys: 1.69 ms, total: 144 ms
          Wall time: 143 ms

Out[112]: -3561.944965714259

In [113]: import sys
          sum([sys.getsizeof(l) for l in mat]) ❹
Out[113]: 215200000
```

❶ 通过嵌套的列表推导来创建矩阵。

❷ 从中提取一些选择过的随机数。

❸ 在列表推导期间首先计算单个列表对象的总和，然后取这些总和的总和。

❹ 加总所有列表对象使用的内存数量。

现在转向 NumPy，看看同一个问题如何解决。方便起见，NumPy 子库 random 提供了多种函数，以初始化 ndarray 对象，同时填入（伪）随机数：

```
In [114]: %time mat = np.random.standard_normal((I, I)) ❶
          CPU times: user 1.01 s, sys: 200 ms, total: 1.21 s
          Wall time: 1.21 s

In [115]: %time mat.sum() ❷
          CPU times: user 29.7 ms, sys: 1.15 ms, total: 30.8 ms
          Wall time: 29.4 ms

Out[115]: -186.12767026606448

In [116]: mat.nbytes ❸
Out[116]: 200000000

In [117]: sys.getsizeof(mat) ❸
Out[117]: 200000112
```

❶ 创建 ndarray 对象，该对象包含标准正态分布的随机数，其速度大约为标准方法的 14 倍。

❷ 计算 ndarray 对象中所有值的总和，其速度大约为标准方法的 4.5 倍。

❸ 由于与数据本身的大小相比，ndarray 对象的内存开销很小，所以 NumPy 方法也能节省一些内存。

> **使用 NumPy 数组**
> 将 NumPy 用于基于数组的操作和算法，通常可以生成比纯 Python 更紧凑、容易理解的代码，并能显著地改进性能。

4.3 NumPy 结构数组

ndarray 类的专门化明显带来了许多真正有价值的好处。但是，过分狭窄的专门化措施可能给大部分基于数组的算法和应用程序造成过大的负担。因此，NumPy 提供结构数组，允许我们至少在每列上使用不同的 dtype。“每列”是什么含义？考虑如下结构

数组对象的初始化：

```
In [118]: dt = np.dtype([('Name', 'S10'), ('Age', 'i4'),
                         ('Height', 'f'), ('Children/Pets', 'i4', 2)]) ❶

In [119]: dt ❶
Out[119]: dtype([('Name', 'S10'), ('Age', '<i4'), ('Height', '<f4'),
           ('Children/Pets', '<i4', (2,))])

In [120]: dt = np.dtype({'names': ['Name', 'Age', 'Height', 'Children/Pets'],
                     'formats':'O int float int,int'.split()}) ❷

In [121]: dt ❷
Out[121]: dtype([('Name', 'O'), ('Age', '<i8'), ('Height', '<f8'),
           ('Children/Pets', [('f0', '<i8'), ('f1', '<i8')])])

In [122]: s = np.array([('Smith', 45, 1.83, (0, 1)),
                        ('Jones', 53, 1.72, (2, 2))], dtype=dt) ❸

In [123]: s ❸
Out[123]: array([('Smith', 45, 1.83, (0, 1)), ('Jones', 53, 1.72, (2, 2))],
          dtype=[('Name', 'O'), ('Age', '<i8'), ('Height', '<f8'),
           ('Children/Pets', [('f0', '<i8'), ('f1', '<i8')])])

In [124]: type(s) ❹
Out[124]: numpy.ndarray
```

❶ 组成复杂的 `dtype`。

❷ 实现相同结果的替代语法

❸ 实例化结构数组，包含两个记录。

❹ 对象类型仍是 `ndarray`。

从某种意义上说，这种构造方法很接近于 SQL 数据库中表的初始化：使用列名、列数据类型，以及某些附加信息（例如，每个字符串对象的最大字符数）。现在，可以简单地用列的名称来访问各个列：

```
In [125]: s['Name'] ❶
Out[125]: array(['Smith', 'Jones'], dtype=object)

In [126]: s['Height'].mean() ❷
Out[126]: 1.775

In [127]: s[0] ❸
Out[127]: ('Smith', 45, 1.83, (0, 1))
```

```
In [128]: s[1]['Age'] ❹
Out[128]: 53
```

❶ 按名称选择一列。

❷ 在选中的列上调用方法。

❸ 选择一个记录。

❹ 选择记录中的一个字段。

总而言之，结构数组是常规 ndarray 对象类型的推广，其中每列的数据类型必须一样，就像 SQL 数据库中的表一样。结构数组的优势之一是某列中的元素可以是另一个多维对象，不一定要遵循基本的 NumPy 数据类型。

> **结构数组**
>
> 除了常规数组之外，NumPy 还提供结构数组。你可以用不同的数据类型甚至结构，按照（命名）列描述和处理相当复杂的面向数组数据结构。它们为 Python 带来了类似于 SQL 表的数据结构，具有常规 ndarray 对象的所有好处（语法、方法、性能）。

4.4 代码向量化

代码的向量化是获得更快执行速度、更紧凑代码的一种策略。基本思路是"一次"在一个复杂对象上进行操作，或者对其应用某个函数，而不是通过在对象的单个元素上循环来进行。在 Python 中，函数式编程工具 map 和 filter 提供了向量化的一些基本手段。在某种意义上，NumPy 的向量化是其核心之一。

4.4.1 基本向量化

正如在 4.3 节中所学到的，简单的数学运算（如计算所有元素的总和）可以直接在 ndarray 对象（通过方法或者通用函数）上进行。更通用的向量化计算也是有可能实施的，例如，我们可以将两个 NumPy 数组按元素相加：

```
In [129]: np.random.seed(100)
          r = np.arange(12).reshape((4, 3)) ❶
          s = np.arange(12).reshape((4, 3)) * 0.5 ❷

In [130]: r ❶
Out[130]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11]])
```

```
In [131]: s ❷
Out[131]: array([[0. , 0.5, 1. ],
                 [1.5, 2. , 2.5],
                 [3. , 3.5, 4. ],
                 [4.5, 5. , 5.5]])

In [132]: r + s ❸
Out[132]: array([[ 0. , 1.5, 3. ],
                 [ 4.5, 6. , 7.5],
                 [ 9. , 10.5, 12. ],
                 [13.5, 15. , 16.5]])
```

❶ 第一个包含随机数的 `ndarray` 对象。

❷ 第二个包含随机数的 `ndarray` 对象。

❸ 以向量化运算（无循环）实施按元素相加。

`NumPy` 还支持所谓的广播，可以在一个操作内组合不同组成的对象。之前我们已经使用过这种方法，考虑如下的示例：

```
In [133]: r + 3 ❶
Out[133]: array([[ 3, 4, 5],
                 [ 6, 7, 8],
                 [ 9, 10, 11],
                 [12, 13, 14]])

In [134]: 2 * r ❷
Out[134]: array([[ 0, 2, 4],
                 [ 6, 8, 10],
                 [12, 14, 16],
                 [18, 20, 22]])

In [135]: 2 * r + 3 ❸
Out[135]: array([[ 3, 5, 7],
                 [ 9, 11, 13],
                 [15, 17, 19],
                 [21, 23, 25]])
```

❶ 在标量加法期间，标量被广播，并加到每个元素上。

❷ 在标量乘法期间，标量也被广播，并乘以每个元素。

❸ 这个线性变换结合了两种运算。

在某种程度上，这些运算也适用于不同组成的 `ndarray` 对象：

```
In [136]: r
Out[136]: array([[ 0, 1, 2],
```

```
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11]])

In [137]: r.shape
Out[137]: (4, 3)

In [138]: s = np.arange(0, 12, 4) ❶
          s ❶
Out[138]: array([0, 4, 8])

In [139]: r + s ❷
Out[139]: array([[ 0,  5, 10],
                [ 3,  8, 13],
                [ 6, 11, 16],
                [ 9, 14, 19]])

In [140]: s = np.arange(0, 12, 3) ❸
          s ❷
Out[140]: array([0, 3, 6, 9])

In [141]: r + s ❹

          ---------------------------------------
          ValueErrorTraceback (most recent call last)
          <ipython-input-141-1890b26ec965> in <module>()
          ----> 1 r + s ❹
          ValueError: operands could not be broadcast together
                      with shapes (4,3) (4,)

In [142]: r.transpose() + s ❺
Out[142]: array([[ 0,  6, 12, 18],
                [ 1,  7, 13, 19],
                [ 2,  8, 14, 20]])

In [143]: sr = s.reshape(-1, 1) ❻
          sr
Out[143]: array([[0],
                [3],
                [6],
                [9]])

In [144]: sr.shape ❻
Out[144]: (4, 1)

In [145]: r + s.reshape(-1, 1) ❻
```

```
Out[145]: array([[ 0,  1,  2],
                 [ 6,  7,  8],
                 [12, 13, 14],
                 [18, 19, 20]])
```

❶ 长度为 3 的新的一维 ndarray 对象。

❷ r（矩阵）和 s（向量）对象可以直接相加。

❸ 另一个长度为 4 的一维 ndarray 对象。

❹ 新的 s（向量）对象的长度与 r 对象的第二维不一致。

❺ 再次转置 r 对象，可以实现向量化加法。

❻ 另外，s 的组成可以改变为（4,1），以实现加法运算（但是，结果不同）。

自定义的 Python 函数往往也可以使用 ndarray 对象。如果实现允许，数组可以像整数或者浮点数对象一样用于函数。考虑如下的函数：

```
In [146]: def f(x):
              return 3 * x + 5 ❶

In [147]: f(0.5) ❷
Out[147]: 6.5

In [148]: f(r) ❸
Out[148]: array([[ 5,  8, 11],
                 [14, 17, 20],
                 [23, 26, 29],
                 [32, 35, 38]])
```

❶ 一个简单的 Python 函数，实现参数为 x 的线性变换

❷ 应用到 Python 浮点数对象的函数 f。

❸ 同一个函数应用到 ndarray 对象，对函数进行向量化和按元素求值。

NumPy 所做的是简单地为对象的各个元素应用函数 f。从这个意义上讲，使用此类操作并没有避免循环，只是在 Python 级别上避免它们，并将循环委托给 NumPy。在 NumPy 级别上，在 ndarray 对象上进行的循环由经过高度优化的代码负责，大部分代码用 C 语言编写，因此一般远快于纯 Python。这就可以解释使用 NumPy 可以解决基于数组用例性能优势的“秘密”了。

4.4.2 内存布局

正如 4.2.2 节中所述，使用 np.zero 初始化 ndarray 对象时，我们为内存布局提供可选的参数。粗略地讲，这个参数指定数组中的哪些元素在相邻的内存中存储。

使用小型数组时，这对数组操作性能没有任何可测的影响。但是，当数组很大时情况就有所不同，这取决于具体在数组上实施的操作。这时内存布局就派上用场了（参见 Eli Bendersky 的文章“Memory Layout of Multi-Dimensional Arrays”）。

为了说明根据内存布局进行的数组处理在科学和金融应用中的重要性，考虑如下多维 ndarray 对象的构造：

```
In [149]: x = np.random.standard_normal((1000000, 5)) ❶

In [150]: y = 2 * x + 3 ❷

In [151]: C = np.array((x, y), order='C') ❸

In [152]: F = np.array((x, y), order='F') ❹

In [153]: x = 0.0; y = 0.0 ❺

In [154]: C[:2].round(2) ❻
Out[154]: array([[[-1.75, 0.34,  1.15, -0.25, 0.98],
                  [ 0.51, 0.22, -1.07, -0.19, 0.26],
                  [-0.46, 0.44, -0.58,  0.82, 0.67],
                  ...,
                  [-0.05,  0.14,  0.17,  0.33,  1.39],
                  [ 1.02,  0.3 , -1.23, -0.68, -0.87],
                  [ 0.83, -0.73,  1.03,  0.34, -0.46]],

                 [[-0.5 , 3.69, 5.31, 2.5 , 4.96],
                  [ 4.03, 3.44, 0.86, 2.62, 3.51],
                  [ 2.08, 3.87, 1.83, 4.63, 4.35],
                  ...,
                  [ 2.9 , 3.28, 3.33, 3.67, 5.78],
                  [ 5.04, 3.6 , 0.54, 1.65, 1.26],
                  [ 4.67, 1.54, 5.06, 3.69, 2.07]]])
```

❶ 在两个维度上有很大对称性的 ndarray 对象。

❷ 原始对象数据的线性变换。

❸ 创建一个 C 顺序（行优先）的二维 ndarray 对象。

❹ 创建一个 F 顺序（列优先）的二维 ndarray 对象。

❺ 内存释放（取决于垃圾收集）。

❻ 来自 C 对象的一些数值。

我们来看看两种 ndarray 对象的基本示例和用例，考虑它们在不同内存布局下的

执行速度：

```
In [155]: %timeit C.sum() ❶
          4.36 ms ± 89.3 µs per loop (mean ± std. dev. of 7 runs, 100
          loops each)

In [156]: %timeit F.sum() ❶
          4.21 ms ± 71.4 µs per loop (mean ± std. dev. of 7 runs, 100
          loops each)

In [157]: %timeit C.sum(axis=0) ❷
          17.9 ms ± 776 µs per loop (mean ± std. dev. of 7 runs, 100
          loops each)

In [158]: %timeit C.sum(axis=1) ❸
          35.1 ms ± 999 µs per loop (mean ± std. dev. of 7 runs, 10
          loops each)

In [159]: %timeit F.sum(axis=0) ❷
          83.8 ms ± 2.63 ms per loop (mean ± std. dev. of 7 runs, 10
          loops each)

In [160]: %timeit F.sum(axis=1) ❸
          67.9 ms ± 5.16 ms per loop (mean ± std. dev. of 7 runs, 10
          loops each)

In [161]: F = 0.0; C = 0.0
```

❶ 计算所有元素的总和。

❷ 计算每行总和（“许多”）。

❸ 计算每列总和（“很少”）。

我们可以将性能结果总结如下：

- 计算所有元素总和时，内存布局不重要；
- 加总 C 顺序的 `ndarray` 对象不管按行和按列都更快（绝对的速度优势）；
- 对于 C 顺序（行优先）的 `ndarray` 对象，按行加总相对快于按列加总；
- 对于 F 顺序（列优先）的 ndarray 对象，按列行加总相对快于按行加总。

4.5 结语

NumPy 是 Python 中用于数值计算的软件包。`ndarray` 类是专门设计的，可以方便、

高效地处理（大型）数值数据。强大的方法和 NumPy 通用方法可以让我们进行代码向量化操作，避免大部分 Python 级别上的缓慢循环。本章介绍的许多方法也适用于 pandas 软件包及其 `DataFrame` 类（参见第 5 章）。

4.6　延伸阅读

NumPy 网站上提供许多有益的资源。

下面是两本 NumPy 的参考用书。

- McKinney, Wes (2017). *Python for Data Analysis*. Sebastopol, CA: O’Reilly.
- VanderPlas, Jake (2016). *Python Data Science Handbook*. Sebastopol, CA: O’Reilly.

第 5 章

pandas 数据分析

数据！数据！数据！没有黏土我无法造出砖来！

——夏洛克•福尔摩斯

本章介绍 pandas，这是一个聚焦于表格数据的数据分析程序库。pandas 是一个强大的工具，它不仅提供了许多实用类和函数，而且很好地封装了来自其他软件包的功能。该工具提供了一个用户接口，能够让用户方便且高效地实现数据分析，特别是金融分析。

本章介绍如下基本数据结构：

对象类型	含义	用途
`DataFrame`	带索引的二维数据对象	按列组织的表格数据
`Series`	带索引的一维数据对象	单一（时间）数据序列

本章的组织如下。

DataFrame 类

本节首先用简单的小数据集来探索 pandas 中 `DataFrame` 类的基本特性和功能；然后说明如何将 NumPy 中的 `ndarray` 对象转换为 `DataFrame` 对象。

基本分析与基本可视化

这两节介绍基本分析与可视化功能（后续的章节将深入介绍这些主题）。

Series 类

本节短小精悍，介绍了 pandas 的 `Series` 类。从某种意义上讲，它是 DataFrame 的一种特例，只有一列数据。

GroupBy 操作

`DataFrame` 的优势之一是根据一列或者多列来分组数据。这一节探索 pandas 的

分组能力。

复杂选择

这一节介绍如何使用（复杂）条件，轻松地从 DataFrame 对象中选择数据。

联接、连接和合并

将不同数据集合而为一是数据分析中的重要操作。pandas 提供了实现这一任务的不同选项，本节将对此进行介绍。

性能特征

pandas 往往提供实现相同目标的多个选项，这符合 Python 的常规。本节简单介绍潜在的性能差异。

5.1 DataFrame 类

pandas（以及本章）的核心是 `DataFrame`，它是用于高效处理表格数据（也就是以列进行组织的数据）的类。为此，`DataFrame` 提供了列标签以及数据集中各行（记录）的索引功能，这与关系数据库的一个表或者 Excel 电子表格类似。

本节介绍 pandas 中 `DataFrame` 类的一些根本特性。该类非常复杂且强大，这里只能介绍其中的一小部分功能。后续章节将提供更多示例，揭示其不同方面的特征。

5.1.1 使用 DataFrame 类的第一步

从最根本的层面上看，`DataFrame` 类用来管理具有索引和标签的数据，这些数据与来自 SQL 数据库表或者电子表格应用中工作表的数据没有太多的不同。考虑如下代码创建的 `DataFrame` 对象：

```
In [1]: import pandas as pd ❶

In [2]: df = pd.DataFrame([10, 20, 30, 40], ❷
                          columns=['numbers'], ❸
                          index=['a', 'b', 'c', 'd']) ❹

In [3]: df ❺
Out[3]:   numbers
        a      10
        b      20
        c      30
        d      40
```

❶ 导入 pandas。

❷ 定义列表对象形式的数据。

❸ 指定列标签。

❹ 指定索引值/标签。

❺ 显示 `DataFrame` 对象的数据以及列和索引标签。

这个简单的例子已经说明了 `DataFrame` 在存储数据上的主要特性，具体如下。

- 数据本身可以用不同组成及类型（列表、元组、ndarray 和字典对象都是候选者）展现。
- 数据以列的方式被组织，可以自定义列名（标签）。
- 索引可以采用不同的格式（如数值、字符串、时间信息）。

使用这种 `DataFrame` 对象总体上相当方便和高效，例如，当你想要进行扩大现有对象等工作时，我推荐你使用 `DataFrame` 对象。相比之下，常规的 `ndarray` 对象更专门化，也更受限制。与此同时，`DataFrame` 对象往往在计算上和 `ndarray` 对象一样高效。下面是简单的例子，说明了 `DataFrame` 对象的典型操作：

```
In [4]: df.index ❶
Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [5]: df.columns ❷
Out[5]: Index(['numbers'], dtype='object')

In [6]: df.loc['c']❸
Out[6]: numbers 30
        Name: c, dtype: int64

In [7]: df.loc[['a', 'd']] ❹
Out[7]:    numbers
        a       10
        d       40

In [8]: df.iloc[1:3] ❺
Out[8]:    numbers
        b 20
        c 30

In [9]: df.sum() ❻
Out[9]: numbers 100
        dtype: int64

In [10]: df.apply(lambda x: x ** 2) ❼
Out[10]:    numbers
         a      100
```

```
         b      400
         c      900
         d      1600

In [11]: df ** 2 ❽
Out[11]:    numbers
         a      100
         b      400
         c      900
         d     1600
```

❶ index 属性和 Index 对象。

❷ columns 属性和 Index 对象。

❸ 选择对应于索引 c 的值。

❹ 选择对应于索引 a 和 b 的两个值。

❺ 通过索引位置选择第二行和第三行。

❻ 计算单列总和。

❼ 使用 apply 方法，以向量化的方式计算平方值。

❽ 和 ndarray 对象一样，直接应用向量化。

与 NumPy 的 ndarray 对象不同，我们可以在两个维度上同时扩增 DataFrame 对象：

```
In [12]: df['floats'] = (1.5, 2.5, 3.5, 4.5) ❶

In [13]: df
Out[13]:    numbers floats
         a       10    1.5
         b       20    2.5
         c       30    3.5
         d       40    4.5

In [14]: df['floats'] ❷
Out[14]: a 1.5
         b 2.5
         c 3.5
         d 4.5
         Name: floats, dtype: float64
```

❶ 添加一个新列，该列包含以元组形式提供的浮点数对象。

❷ 选择该列并显示其数据和索引标签。

也可以使用整个 `DataFrame` 对象来定义一个新列。在这种情况下，索引自动对齐：

```
In [15]: df['names'] = pd.DataFrame(['Yves', 'Sandra', 'Lilli', 'Henry'],
                                    index=['d', 'a', 'b', 'c']) ❶

In [16]: df
Out[16]:   numbers floats    names
         a      10   1.5     Sandra
         b      20   2.5     Lilli
         c      30   3.5     Henry
         d      40   4.5     Yves
```

❶ 根据一个 `DataFrame` 对象创建另一个新列。

附加数据的方法也类似。但是，在下面的例子中，我们会看到在操作过程中必须避免的一个副作用——索引被简单的编号索引代替：

```
In [17]: df.append({'numbers': 100, 'floats': 5.75, 'names': 'Jil'},
                        ignore_index=True) ❶
Out[17]:    numbers  floats   names
         0       10    1.50  Sandra
         1       20    2.50   Lilli
         2       30    3.50   Henry
         3       40    4.50    Yves
         4       100    5.75    Jil

In [18]: df = df.append(pd.DataFrame({'numbers': 100, 'floats': 5.75,
                                      'names': 'Jil'}, index=['y',])) ❷

In [19]: df
Out[19]:    numbers floats    names
         a       10   1.50   Sandra
         b       20   2.50    Lilli
         c       30   3.50    Henry
         d       40   4.50     Yves
         y       100  5.75      Jil

In [20]: df = df.append(pd.DataFrame({'names': 'Liz'}, index=['z',]),
                        sort=False) ❸
In [21]: df
Out[21]:    numbers floats    names
         a     10.0   1.50   Sandra
         b     20.0   2.50    Lilli
         c     30.0   3.50    Henry
         d     40.0   4.50     Yves
         y     100.0  5.75      Jil
```

```
          z     NaN     NaN      Liz

In [22]: df.dtypes ❹
Out[22]: numbers float64
         floats  float64
         names     object
         dtype:object
```

❶ 通过一个字典对象附加新行；这是一个临时操作，操作期间索引信息丢失。

❷ 根据带索引信息的 DataFrame 对象附加行；原始索引信息保留。

❸ 向 DataFrame 对象附加不完整的数据行，生成 NaN 值。

❹ 返回各列的不同 dtype，这与结构化 ndarray 对象类似。

尽管此时有遗漏的值，但大部分的方法调用仍然可以正常工作：

```
In [23]: df[['numbers', 'floats']].mean() ❶
Out[23]: numbers    40.00
         Floats      3.55
         dtype: float64

In [24]: df[['numbers', 'floats']].std() ❷
Out[24]: numbers    35.355339
         Floats      1.662077
         dtype: float64
```

❶ 计算指定两列的均值（忽略带 NaN 值的行）。

❷ 计算指定两列的标准差（忽略带 NaN 值的行）。

5.1.2　使用 DataFrame 类的第二步

本节的例子基于一个包含标准正态分布随机数的 ndarray 对象。该例探索管理时间序列数据的更多功能，如 DatetimeIndex：

```
In [25]: import numpy as np

In [26]: np.random.seed(100)

In [27]: a = np.random.standard_normal((9, 4))

In [28]: a
Out[28]: array([[-1.74976547, 0.3426804 ,  1.1530358 , -0.25243604],
                [ 0.98132079, 0.51421884,  0.22117967, -1.07004333],
                [-0.18949583, 0.25500144, -0.45802699,  0.43516349],
                [-0.58359505, 0.81684707,  0.67272081, -0.10441114],
```

```
                      [-0.53128038, 1.02973269, -0.43813562, -1.11831825],
                      [ 1.61898166, 1.54160517, -0.25187914, -0.84243574],
                      [ 0.18451869, 0.9370822 ,  0.73100034,  1.36155613],
                      [-0.32623806, 0.05567601,  0.22239961, -1.443217  ],
                      [-0.75635231, 0.81645401,  0.75044476, -0.45594693]])
```

你可以更直接地构造 `DataFrame` 对象（正如前面所见），但是使用 `ndarray` 对象通常是一个更好的选择，因为 pandas 将保留基本结构，"只"添加元信息（例如索引值）。这也代表了金融应用和科学研究的一种典型用例。例如：

```
In [29]: df = pd.DataFrame(a) ❶

In [30]: df
Out[30]:          0         1         2         3
         0 -1.749765 0.342680  1.153036 -0.252436
         1  0.981321 0.514219  0.221180 -1.070043
         2 -0.189496 0.255001 -0.458027  0.435163
         3 -0.583595 0.816847  0.672721 -0.104411
         4 -0.531280 1.029733 -0.438136 -1.118318
         5  1.618982 1.541605 -0.251879 -0.842436
         6  0.184519 0.937082  0.731000  1.361556
         7 -0.326238 0.055676  0.222400 -1.443217
         8 -0.756352 0.816454  0.750445 -0.455947
```

❶ 由 `ndarray` 对象创建一个 `DataFrame` 对象。

表 5-1 列出了 `DataFrame()` 函数使用的参数。表中，"类似数组"意味着与 `ndarray` 对象类似的数据结构——如列表对象。"索引"是 pandas 的 `Index` 类的一个实例。

表 5-1　DataFrame()函数参数

参数	格式	描述
`data`	`ndarray/dict/DataFrame`	`DataFrame` 数据；`dict` 可以包含序列、`ndarray` 和列表
`index`	索引/类似数组	使用的索引；默认为 `range(n)`
`columns`	索引/类似数组	使用的列标题；默认为 `range(n)`
`dtype`	`dtype`，默认 `None`	使用/强制的数据类型；否则通过推导得出
`copy`	布尔值，默认 `None`	从输入复制数据

和结构数组一样，我们已经了解，`DataFrame` 对象可以通过指定一个具有合适数量元素的列表来直接定义列名。下面的例子说明，我们可以随时定义/更改 `DataFrame` 对象的属性：

```
In [31]: df.columns = ['No1', 'No2', 'No3', 'No4'] ❶

In [32]: df
```

```
Out[32]:          No1       No2        No3        No4
         0 -1.749765 0.342680  1.153036 -0.252436
         1  0.981321 0.514219  0.221180 -1.070043
         2 -0.189496 0.255001 -0.458027  0.435163
         3 -0.583595 0.816847  0.672721 -0.104411
         4 -0.531280 1.029733 -0.438136 -1.118318
         5  1.618982 1.541605 -0.251879 -0.842436
         6  0.184519 0.937082  0.731000  1.361556
         7 -0.326238 0.055676  0.222400 -1.443217
         8 -0.756352 0.816454  0.750445 -0.455947
In [33]: df['No2'].mean() ❷
Out[33]: 0.7010330941456459
```

❶ 通过列表对象指定列标签。

❷ 选择一列很容易。

为了高效处理金融事件序列数据，我们还必须很好地处理时间索引。这也可以视为 pandas 的一个重要优势。例如，假定分为 4 列的 9 个数据项对应于 2019 年 1 月开始的月底数据。然后，用 `date_range()`生成一个 `DatetimeIndex` 对象：

```
In [34]: dates = pd.date_range('2019-1-1', periods=9, freq='M') ❶

In [35]: dates
Out[35]: DatetimeIndex(['2019-01-31', '2019-02-28', '2019-03-31', '2019-04-30',
                        '2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
                       '2019-09-30'],
                        dtype='datetime64[ns]', freq='M')
```

❶ 创建一个 `DatetimeIndex` 对象。

表 5-2 列出了 `date_range()`函数的参数。

表 5-2　**`date_range()`函数参数**

参数	格式	描述
`start`	字符串/日期时间	生成日期的左界
`end`	字符串/日期时间	生成日期的右界
`periods`	整数/None	期数（如果 `start` 或者 `end` 空缺）
`freq`	字符串/日期偏移	频率字符串，例如 5D（5 天）
`tz`	字符串/None	本地化索引的时区名称
`normalize`	布尔值，默认 None	将 `start` 和 `end` 规范化为午夜
`name`	字符串，默认 None	结果索引名称

下列代码将刚刚创建的 `DatetimeIndex` 对象定义为相关的索引对象，并建立原始数据集的一个时间序列：

```
In [36]: df.index = dates

In [37]: df
Out[37]:                  No1      No2       No3       No4
         2019-01-31 -1.749765 0.342680  1.153036 -0.252436
         2019-02-28  0.981321 0.514219  0.221180 -1.070043
         2019-03-31 -0.189496 0.255001 -0.458027  0.435163
         2019-04-30 -0.583595 0.816847  0.672721 -0.104411
         2019-05-31 -0.531280 1.029733 -0.438136 -1.118318
         2019-06-30  1.618982 1.541605 -0.251879 -0.842436
         2019-07-31  0.184519 0.937082  0.731000  1.361556
         2019-08-31 -0.326238 0.055676  0.222400 -1.443217
         2019-09-30 -0.756352 0.816454  0.750445 -0.455947
```

至于在 `date_range()`函数帮助下生成的 `DatetimeIndex` 对象，频率参数 `freq` 有多种选择。表 5-3 列出了其所有选项。

表 5-3　**`date_range()`函数的频率参数值**

别名	描述
B	交易日
C	自定义交易日（试验性）
D	日历日
W	每周
M	每月底
BM	每月最后一个交易日
MS	月初
BMS	每月第一个交易日
Q	季度末
BQ	每季度最后一个交易日
QS	季度初
BQS	每季度第一个交易日
A	每年底
BA	每年最后一个交易日
AS	每年初
BAS	每年第一个交易日

续表

别名	描述
H	每小时
T	每分钟
S	每秒
L	毫秒
U	微秒

在某些情况下，以 `ndarray` 对象的形式访问原始数据是有好处的。values 属性提供直接访问的方式：

```
In [38]: df.values
Out[38]: array([[-1.74976547, 0.3426804 , 1.1530358 , -0.25243604],
                [ 0.98132079, 0.51421884, 0.22117967, -1.07004333],
                [-0.18949583, 0.25500144, -0.45802699, 0.43516349],
                [-0.58359505, 0.81684707, 0.67272081, -0.10441114],
                [-0.53128038, 1.02973269, -0.43813562, -1.11831825],
                [ 1.61898166, 1.54160517, -0.25187914, -0.84243574],
                [ 0.18451869, 0.9370822 , 0.73100034, 1.36155613],
                [-0.32623806, 0.05567601, 0.22239961, -1.443217 ],
                [-0.75635231, 0.81645401, 0.75044476, -0.45594693]])

In [39]: np.array(df)
Out[39]: array([[-1.74976547, 0.3426804 , 1.1530358 , -0.25243604],
                [ 0.98132079, 0.51421884, 0.22117967, -1.07004333],
                [-0.18949583, 0.25500144, -0.45802699, 0.43516349],
                [-0.58359505, 0.81684707, 0.67272081, -0.10441114],
                [-0.53128038, 1.02973269, -0.43813562, -1.11831825],
                [ 1.61898166, 1.54160517, -0.25187914, -0.84243574],
                [ 0.18451869, 0.9370822 , 0.73100034, 1.36155613],
                [-0.32623806, 0.05567601, 0.22239961, -1.443217 ],
                [-0.75635231, 0.81645401, 0.75044476, -0.45594693]])
```

数组和 DataFrame

通常可以从一个 `ndarray` 对象生成 `DataFrame` 对象。也可以通过 NumPy 的 `np.array` 函数或 `DataFrame` 类的 `values` 属性，来从 DataFrame 对象生成 ndarray 对象。

5.2 基本分析

和 NumPy 的 `ndarray` 类一样，pandas 的 `DataFrame` 类有多个便利的内建方法。首

先考虑 info()和 describe()方法：

```
In [40]: df.info() ❶
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 9 entries, 2019-01-31 to 2019-09-30
         Freq: M
         Data columns (total 4 columns):
         No1    9 non-null float64
         No2    9 non-null float64
         No3    9 non-null float64
         No4    9 non-null float64
         dtypes: float64(4)
         memory usage: 360.0 bytes

In [41]: df.describe() ❷
Out[41]:             No1      No2       No3       No4
         Count  9.000000 9.000000  9.000000  9.000000
         Mean  -0.150212 0.701033  0.289193 -0.387788
         Std    0.988306 0.457685  0.579920  0.877532
         Min   -1.749765 0.055676 -0.458027 -1.443217
         25%   -0.583595 0.342680 -0.251879 -1.070043
         50%   -0.326238 0.816454  0.222400 -0.455947
         75%    0.184519 0.937082  0.731000 -0.104411
         max    1.618982 1.541605  1.153036  1.361556
```

❶ 提供关于数据、列和索引的元信息。

❷ 提供有用的每列汇总统计（对于数值数据）信息。

此外，你可以轻松地求得按列/行计算的总和、平均值和累计总和：

```
In [43]: df.sum() ❶
Out[43]: No1   -1.351906
         No2    6.309298
         No3    2.602739
         No4   -3.490089
         dtype: float64

In [44]: df.mean() ❷
Out[44]: No1   -0.150212
         No2    0.701033
         No3    0.289193
         No4   -0.387788
         dtype: float64

In [45]: df.mean(axis=0) ❷
Out[45]: No1   -0.150212
```

```
         No2     0.701033
         No3     0.289193
         No4    -0.387788
         dtype: float64

In [46]: df.mean(axis=1) ❸
Out[46]: 2019-01-31   -0.126621
         2019-02-28    0.161669
         2019-03-31    0.010661
         2019-04-30    0.200390
         2019-05-31   -0.264500
         2019-06-30    0.516568
         2019-07-31    0.803539
         2019-08-31   -0.372845
         2019-09-30    0.088650
         Freq: M, dtype: float64

In [47]: df.cumsum() ❹
Out[47]:                   No1       No2       No3        No4
         2019-01-31 -1.749765 0.342680 1.153036 -0.252436
         2019-02-28 -0.768445 0.856899 1.374215 -1.322479
         2019-03-31 -0.957941 1.111901 0.916188 -0.887316
         2019-04-30 -1.541536 1.928748 1.588909 -0.991727
         2019-05-31 -2.072816 2.958480 1.150774 -2.110045
         2019-06-30 -0.453834 4.500086 0.898895 -2.952481
         2019-07-31 -0.269316 5.437168 1.629895 -1.590925
         2019-08-31 -0.595554 5.492844 1.852294 -3.034142
         2019-09-30 -1.351906 6.309298 2.602739 -3.490089
```

❶ 列总和。

❷ 列均值。

❸ 行均值。

❹ 列累计总和（从第一个索引位置起）。

`DataFrame` 对象也能理解为 NumPy 通用函数，这与预期相符：

```
In [48]: np.mean(df) ❶
Out[48]: No1    -0.150212
         No2     0.701033
         No3     0.289193
         No4    -0.387788
         dtype: float64

In [49]: np.log(df) ❷
```

```
Out[49]:                   No1       No2       No3       No4
         2019-01-31       NaN -1.070957  0.142398       NaN
         2019-02-28 -0.018856 -0.665106 -1.508780       NaN
         2019-03-31       NaN -1.366486       NaN -0.832033
         2019-04-30       NaN -0.202303 -0.396425       NaN
         2019-05-31       NaN  0.029299       NaN       NaN
         2019-06-30  0.481797  0.432824       NaN       NaN
         2019-07-31 -1.690005 -0.064984 -0.313341  0.308628
         2019-08-31       NaN -2.888206 -1.503279       NaN
         2019-09-30       NaN -0.202785 -0.287089       NaN

In [50]: np.sqrt(abs(df)) ❸
Out[50]:                   No1       No2       No3       No4
         2019-01-31  1.322787  0.585389  1.073795  0.502430
         2019-02-28  0.990616  0.717091  0.470297  1.034429
         2019-03-31  0.435311  0.504977  0.676777  0.659669
         2019-04-30  0.763934  0.903796  0.820196  0.323127
         2019-05-31  0.728890  1.014757  0.661918  1.057506
         2019-06-30  1.272392  1.241614  0.501876  0.917843
         2019-07-31  0.429556  0.968030  0.854986  1.166857
         2019-08-31  0.571173  0.235958  0.471593  1.201340
         2019-09-30  0.869685  0.903578  0.866282  0.675238

In [51]: np.sqrt(abs(df)).sum() ❹
Out[51]: No1    7.384345
         No2    7.075190
         No3    6.397719
         No4    7.538440
         dtype: float64

In [52]: 100 * df + 100 ❺
Out[52]:                    No1         No2         No3        No4
         2019-01-31 -74.976547 134.268040 215.303580 74.756396
         2019-02-28 198.132079 151.421884 122.117967 -7.004333
         2019-03-31  81.050417 125.500144  54.197301 143.516349
         2019-04-30  41.640495 181.684707 167.272081  89.558886
         2019-05-31  46.871962 202.973269  56.186438 -11.831825
         2019-06-30 261.898166 254.160517  74.812086  15.756426
         2019-07-31 118.451869 193.708220 173.100034 236.155613
         2019-08-31  67.376194 105.567601 122.239961 -44.321700
         2019-09-30  24.364769 181.645401 175.044476  54.405307
```

❶ 列均值。

❷ 每个元素的自然对数；显示警告信息，但计算继续进行，得到多个 NaN 值。

❸ 每个元素绝对值的平方根。

❹ 按列求均值。

❺ 数值数据的线性变换。



NumPy 通用函数

一般来说，在 NumPy 通用函数适用于 `ndarray` 对象的所有情况下，都可以将这些函数应用到包含相同数据的 pandas 的 `DataFrame` 对象。

pandas 有相当强的容错能力，它可以捕捉错误，在对应数学运算失败时放入 `NaN` 值。不仅如此，正如前面已经展示的，在许多情况下，它还可以将这些不完整的数据集当成完整数据集来使用。这种做法很方便，因为现实中不完整数据集往往比人们预想的要多。

5.3 基本可视化

如果数据存储在 `DataFrame` 对象中，那么数据图表的绘制通常只需要一行代码（见图 5-1）。

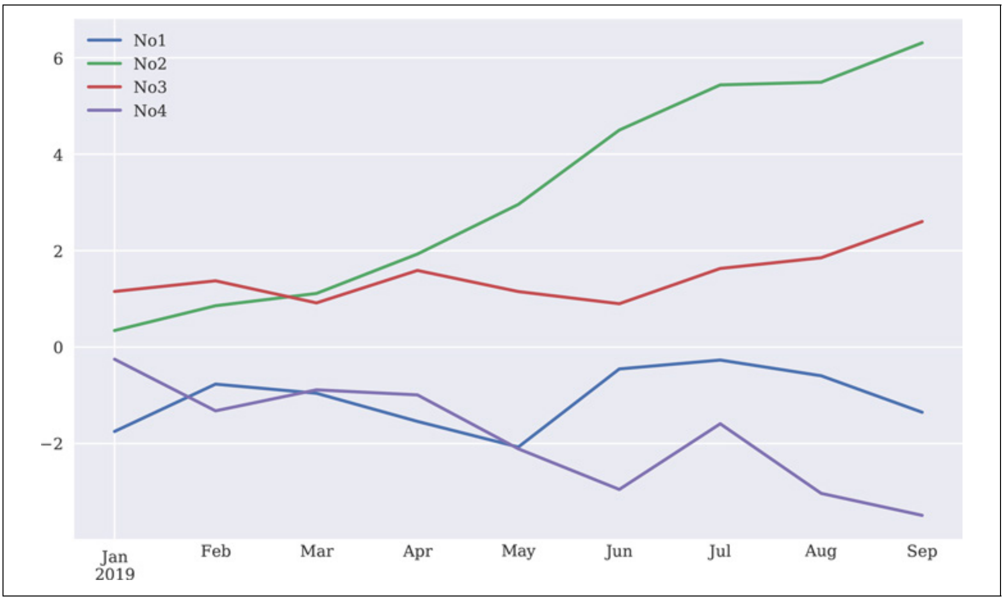


图 5-1　**`DataFrame` 对象的折线图**

```
In [53]: from pylab import plt, mpl ❶
         plt.style.use('seaborn') ❶
         mpl.rcParams['font.family'] = 'serif' ❶
         %matplotlib inline

In [54]: df.cumsum().plot(lw=2.0, figsize=(10, 6)); ❷
```

❶ 定制绘图样式。

❷ 绘制 4 列累计总和的折线图。

基本上，pandas 提供了专为 `DataFrame` 对象设计的 `matplotlib`（参见第 7 章）包装器。`plot()`方法的参数如表 5-4 所示。

表 5-4　plot()方法参数

参数	格式	描述
`x`	标签/位置，默认 `None`	只在列值为 *x* 刻度时使用
`y`	标签/位置，默认 `None`	只在列值为 *x* 刻度时使用
`subplots`	布尔值，默认 `False`	子图中的绘图列
`sharex`	布尔值，默认 `True`	共用 *x* 轴
`sharey`	布尔值，默认 `False`	共用 *y* 轴
`use_index`	布尔值，默认 `True`	使用 `DataFrame.index` 作为 *x* 轴刻度
`stacked`	布尔值，默认 `False`	堆叠（只用于柱状图）
`sort_columns`	布尔值，默认 `False`	在绘图之前将列按字母顺序排列
`title`	字符串，默认 `None`	图表标题
`grid`	布尔值，默认 `False`	水平和垂直网格线
`legend`	布尔值，默认 `True`	标签图例
`ax`	`matplotlib` 轴对象	绘图使用的 matplotlib 轴对象
`style`	字符串或者列表/字典	绘图线形（每列）
`kind`	`line/bar/barh/kde/density`	图表类型
`logx`	布尔值，默认 `False`	*x* 轴的对数刻度
`logy`	布尔值，默认 `False`	*y* 轴的对数刻度
`xticks`	序列，默认 `Index`	*x* 轴刻度
`yticks`	序列，默认 `Values`	*y* 轴刻度
`xlim`	二元组，列表	*x* 轴界限
`ylim`	二元组，列表	*y* 轴界限
`rot`	整数，默认 `None`	旋转 *x* 刻度
`secondary_y`	布尔值/序列，默认 `False`	第二个 *y* 轴
`mark_right`	布尔值，默认 `True`	第二个 *y* 轴自动设置标签
`colormap`	字符串/颜色映射对象，默认 `None`	用于绘图的颜色映射
`kwds`	关键字	传递给 matplotlib 选项

再举个例子，考虑相同数据的柱状图（见图 5-2）：

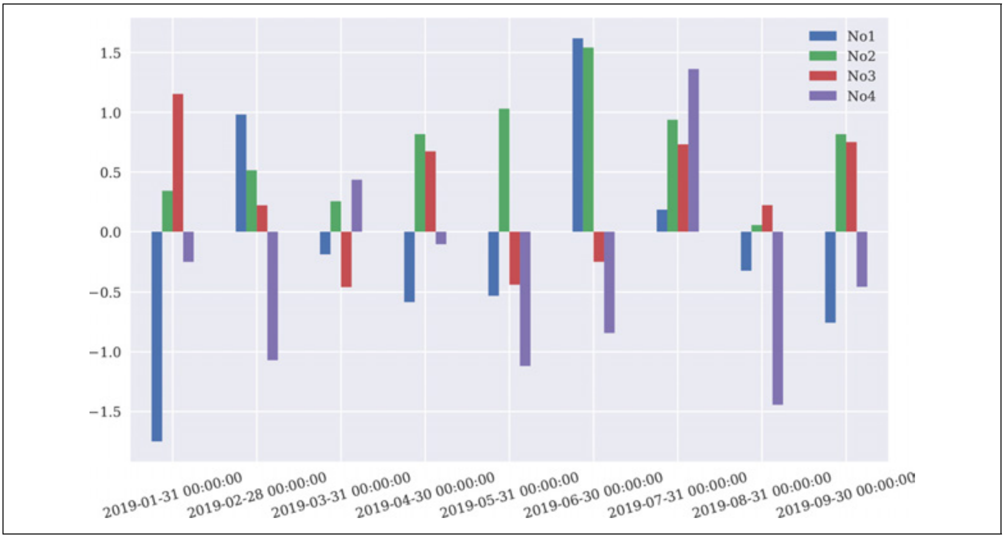


图 5-2　DataFrame 对象的柱状图

```
In [55]: df.plot.bar(figsize=(10, 6), rot=15); ❶
         # df.plot(kind='bar', figsize=(10, 6)) ❷
```

❶ plot.bar()绘制柱状图。

❷ 替代语法：使用 kind 参数改变绘图类型。

5.4　Series 类

迄今为止，本章主要介绍了 pandas 的 DataFrame 类。Series 是 pandas 自带的另一个重要的类。它的特点是只有一列数据。从这个意义上讲，它是 DataFrame 类的特例，两者之间有许多共同特性和功能，但不完全相同。从多列的 DataFrame 对象上选取一列，可以得到 Series 对象：

```
In [56]: type(df)
Out[56]: pandas.core.frame.DataFrame

In [57]: S = pd.Series(np.linspace(0, 15, 7), name='series')

In [58]: S
Out[58]: 0     0.0
         1     2.5
         2     5.0
         3     7.5
         4    10.0
         5    12.5
         6    15.0
```

```
         Name: series, dtype: float64

In [59]: type(S)
Out[59]: pandas.core.series.Series

In [60]: s = df['No1']

In [61]: s
Out[61]: 2019-01-31 -1.749765
         2019-02-28  0.981321
         2019-03-31 -0.189496
         2019-04-30 -0.583595
         2019-05-31 -0.531280
         2019-06-30  1.618982
         2019-07-31  0.184519
         2019-08-31 -0.326238
         2019-09-30 -0.756352
         Freq: M, Name: No1, dtype: float64

In [62]: type(s)
Out[62]: pandas.core.series.Series
```

DataFrame 的主要方法也可用于 Series 对象，我们以 mean 和 plot 方法为例（见图 5-3）：

```
In [63]: s.mean()
Out[63]: -0.15021177307319458

In [64]: s.plot(lw=2.0, figsize=(10, 6));
```

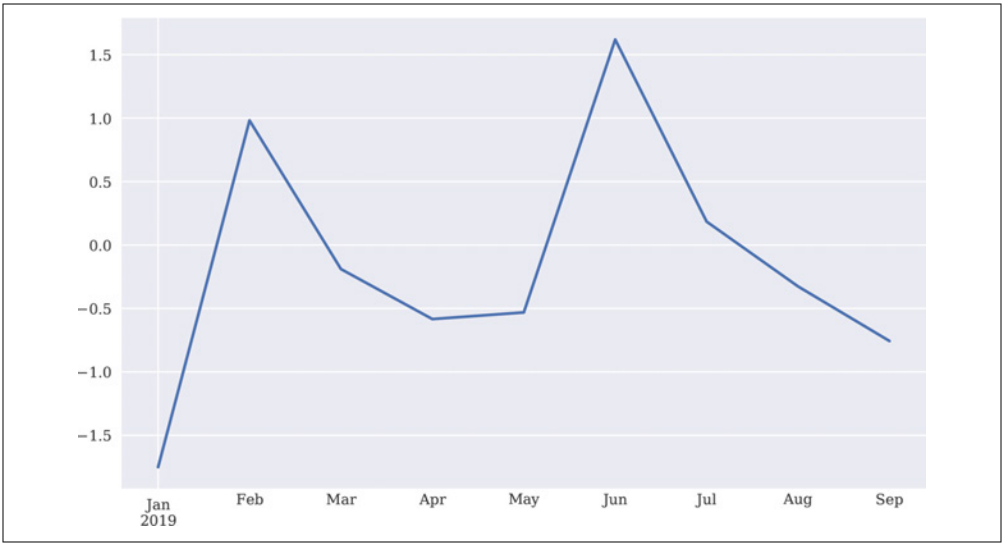


图 5-3　Series 对象的折线图

5.5 GroupBy 操作

pandas 具备强大而灵活的分组功能，工作方式类似于 SQL 中的分组和 Microsoft Excel 中的透视表。为了进行分组，我们添加一列，表示对应索引数据所属的季度：

```
In [65]: df['Quarter'] = ['Q1', 'Q1', 'Q1', 'Q2', 'Q2',
                          'Q2', 'Q3', 'Q3', 'Q3']
         df
Out[65]:                   No1       No2       No3       No4 Quarter
         2019-01-31 -1.749765  0.342680  1.153036 -0.252436      Q1
         2019-02-28  0.981321  0.514219  0.221180 -1.070043      Q1
         2019-03-31 -0.189496  0.255001 -0.458027  0.435163      Q1
         2019-04-30 -0.583595  0.816847  0.672721 -0.104411      Q2
         2019-05-31 -0.531280  1.029733 -0.438136 -1.118318      Q2
         2019-06-30  1.618982  1.541605 -0.251879 -0.842436      Q2
         2019-07-31  0.184519  0.937082  0.731000  1.361556      Q3
         2019-08-31 -0.326238  0.055676  0.222400 -1.443217      Q3
         2019-09-30 -0.756352  0.816454  0.750445 -0.455947      Q3
```

现在，我们可以根据 `Quarter` 列分组，并输出单独组的统计量：

```
In [66]: groups = df.groupby('Quarter') ❶

In [67]: groups.size() ❷
Out[67]: Quarter
         Q1    3
         Q2    3
         Q3    3
         dtype: int64

In [68]: groups.mean() ❸
Out[68]:               No1      No2       No3       No4
Quarter
         Q1      -0.319314 0.370634  0.305396 -0.295772
         Q2       0.168035 1.129395 -0.005765 -0.688388
         Q3      -0.299357 0.603071  0.567948 -0.179203

In [69]: groups.max() ❹
Out[69]:               No1      No2      No3       No4
         Quarter
         Q1       0.981321 0.514219 1.153036  0.435163
         Q2       1.618982 1.541605 0.672721 -0.104411
         Q3       0.184519 0.937082 0.750445  1.361556

In [70]: groups.aggregate([min, max]).round(2) ❺
```

```
Out[70]:            No1       No2        No3        No4
                    Min  max  min  max   min  max   min   max
         Quarter
         Q1      -1.75 0.98 0.26 0.51 -0.46 1.15 -1.07  0.44
         Q2      -0.58 1.62 0.82 1.54 -0.44 0.67 -1.12 -0.10
         Q3      -0.76 0.18 0.06 0.94  0.22 0.75 -1.44  1.36
```

❶ 根据 `Quarter` 列分组。

❷ 给出每个分组的行数。

❸ 给出每列的均值。

❹ 给出每列的最大值。

❺ 给出每列的最小值和最大值。

分组也可以在多列上进行。为此，我们添加另一列，表示索引日期的月份是奇数还是偶数：

```
In [71]: df['Odd_Even'] = ['Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even',
                           'Odd', 'Even', 'Odd']

In [72]: groups = df.groupby(['Quarter', 'Odd_Even'])

In [73]: groups.size()
Out[73]: Quarter  Odd_Even
         Q1       Even        1
                  Odd         2
         Q2       Even        2
                  Odd         1
         Q3       Even        1
                  Odd         2
         dtype: int64

In [74]: groups[['No1', 'No4']].aggregate([sum, np.mean])
Out[74]:                        No1                 No4
                                Sum      mean       sum      mean
         Quarter Odd_Even
         Q1      Even      0.981321  0.981321 -1.070043 -1.070043
                 Odd      -1.939261 -0.969631  0.182727  0.091364
         Q2      Even      1.035387  0.517693 -0.946847 -0.473423
                 Odd      -0.531280 -0.531280 -1.118318 -1.118318
         Q3      Even     -0.326238 -0.326238 -1.443217 -1.443217
                 Odd      -0.571834 -0.285917  0.905609  0.452805
```

对 pandas 和 `DataFrame` 对象使用的介绍到此结束，下面将这组工具应用到现实世界的金融数据中。

5.6 复杂选择

数据选择往往是通过列值上的条件公式实现的，也可以通过符合逻辑的方式来组合多个条件。考虑如下数据集。

```
In [75]: data = np.random.standard_normal((10, 2)) ❶

In [76]: df = pd.DataFrame(data, columns=['x', 'y']) ❷

In [77]: df.info() ❷
         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 10 entries, 0 to 9
         Data columns (total 2 columns):
         X    10 non-null float64
         Y    10 non-null float64
         dtypes: float64(2)
         memory usage: 240.0 bytes

In [78]: df.head() ❸
Out[78]:          x         y
         0  1.189622 -1.690617
         1 -1.356399 -1.232435
         2 -0.544439 -0.668172
         3  0.007315 -0.612939
         4  1.299748 -1.733096

In [79]: df.tail() ❹
Out[79]:          x         y
         5 -0.983310  0.357508
         6 -1.613579  1.470714
         7 -1.188018 -0.549746
         8 -0.940046 -0.827932
         9  0.108863  0.507810
```

❶ 包含标准正态分布随机数的 `ndarray` 对象。

❷ 包含相同随机数的 `DataFrame` 对象。

❸ 通过 `head` 方法取得前 5 行。

❹ 通过 `tail` 方法取得最后 5 行。

下面的代码说明了 Python 比较运算符和逻辑运算符在两列值上的应用：

```
In [80]: df['x'] > 0.5 ❶
```

```
Out[80]: 0     True
         1    False
         2    False
         3    False
         4     True
         5    False
         6    False
         7    False
         8    False
         9    False
         Name: x, dtype: bool

In [81]: (df['x'] > 0) & (df['y'] < 0) ❷
Out[81]: 0     True
         1    False
         2    False
         3     True
         4     True
         5    False
         6    False
         7    False
         8    False
         9    False
         dtype: bool

In [82]: (df['x'] > 0) | (df['y'] < 0) ❸
Out[82]: 0     True
         1     True
         2     True
         3     True
         4     True
         5    False
         6    False
         7     True
         8     True
         9     True
         dtype: bool
```

❶ 检查 `x` 列的值是否大于 0.5。

❷ 检查是否 `x` 列的值为正且 `y` 列的值为负。

❸ 检查是否 `x` 列的值为正或 `y` 列的值为负。

利用得到的布尔型 `Series` 对象，就可以很容易地实现复杂数据（行）的选择。另外，也可以使用 `query` 方法并以字符串对象的形式来传递条件：

```
In [83]: df[df['x'] > 0] ❶
Out[83]:          x         y
         0  1.189622 -1.690617
         3  0.007315 -0.612939
         4  1.299748 -1.733096
         9  0.108863  0.507810

In [84]: df.query('x > 0') ❶
Out[84]:          x         y
         0  1.189622 -1.690617
         3  0.007315 -0.612939
         4  1.299748 -1.733096
         9  0.108863  0.507810

In [85]: df[(df['x'] > 0) & (df['y'] < 0)] ❷
Out[85]:          x         y
         0  1.189622 -1.690617
         3  0.007315 -0.612939
         4  1.299748 -1.733096

In [86]: df.query('x > 0 & y < 0') ❷
Out[86]:          x         y
         0  1.189622 -1.690617
         3  0.007315 -0.612939
         4  1.299748 -1.733096

In [87]: df[(df.x > 0) | (df.y < 0)] ❸
Out[87]:          x         y
         0  1.189622 -1.690617
         1 -1.356399 -1.232435
         2 -0.544439 -0.668172
         3  0.007315 -0.612939
         4  1.299748 -1.733096
         7 -1.188018 -0.549746
         8 -0.940046 -0.827932
         9  0.108863  0.507810
```

❶ 所有第 x 列的值大于 0.5 的行。

❷ 所有第 x 列的值为正且 y 的列值为负的行。

❸ 所有第 x 列的值为正或 y 的列值为负的行（各列通过对应属性访问）。

比较运算符也可以一次性应用到整个 `DataFrame` 对象上：

```
In [88]: df > 0 ❶
Out[88]:       x      y
```

```
        0   True False
        1  False False
        2  False False
        3   True False
        4   True False
        5  False  True
        6  False  True
        7  False False
        8  False False
        9   True  True

In [89]: df[df > 0] ❷
Out[89]:           x         y
        0  1.189622       NaN
        1       NaN       NaN
        2       NaN       NaN
        3  0.007315       NaN
        4  1.299748       NaN
        5       NaN  0.357508
        6       NaN  1.470714
        7       NaN       NaN
        8       NaN       NaN
        9  0.108863  0.507810
```

❶ `DataFrame` 对象中的哪些值为正？

❷ 选择所有满足要求的值，并将其他值设为 `NaN`。

5.7 联接、连接和合并

本节简单介绍连接 `DataFrame` 对象形式的两个简单数据集的不同方法。这两个简单数据集如下所示：

```
In [90]: df1 = pd.DataFrame(['100', '200', '300', '400'],
                                index=['a', 'b', 'c', 'd'],
                                columns=['A',])

In [91]: df1
Out[91]:      A
        a   100
        b   200
        c   300
        d   400

In [92]: df2 = pd.DataFrame(['200', '150', '50'],
```

```
                                   index=['f', 'b', 'd'],
                                   columns=['B',])

In [93]: df2
Out[93]:      B
         f 200
         b 150
         d 50
```

5.7.1 联接

联接（Concatenation）或者附加（Appending）本质上指的是将一个 `DataFrame` 对象中的行添加到另一个 `DataFrame` 对象上，这可通过 `append` 方法或者 `pd.concat` 函数完成。需要认真考虑的是索引值的处理方法：

```
In [94]: df1.append(df2, sort=False) ❶
Out[94]:      A    B
         a  100  NaN
         b  200  NaN
         c  300  NaN
         d  400  NaN
         f  NaN  200
         b  NaN  150
         d  NaN   50

In [95]: df1.append(df2, ignore_index=True, sort=False) ❷
Out[95]:      A    B
         0  100  NaN
         1  200  NaN
         2  300  NaN
         3  400  NaN
         4  NaN  200
         5  NaN  150
         6  NaN   50

In [96]: pd.concat((df1, df2), sort=False) ❸
Out[96]:      A    B
         a  100  NaN
         b  200  NaN
         c  300  NaN
         d  400  NaN
         f  NaN  200
         b  NaN  150
         d  NaN   50

In [97]: pd.concat((df1, df2), ignore_index=True, sort=False) ❹
```

```
Out[97]:       A    B
         0  100  NaN
         1  200  NaN
         2  300  NaN
         3  400  NaN
         4  NaN  200
         5  NaN  150
         6  NaN   50
```

❶ 将 `df2` 的数据附加到 `df1` 中作为新行。

❷ 完成同样的工作，但忽略索引。

❸ 与第一个附加操作效果相同。

❹ 与第二个附加操作效果相同。

5.7.2　连接

连接两个数据集时，`DataFrame` 对象的顺序很重要。只有第一个 `DataFrame` 对象的索引值被使用，这种默认行为称为左连接（left join）：

```
In [98]: df1.join(df2) ❶
Out[98]:       A    B
         a  100  NaN
         b  200  150
         c  300  NaN
         d  400   50

In [99]: df2.join(df1) ❷
Out[99]:       B    A
         f  200  NaN
         b  150  200
         d   50  400
```

❶ `df1` 的索引值有意义。

❷ `df2` 的索引值有意义。

共有 4 种不同的连接方法，每种方法都导致索引值和对应数据行的不同处理行为：

```
In [100]: df1.join(df2, how='left') ❶
Out[100]:       A    B
          a  100  NaN
          b  200  150
          c  300  NaN
          d  400   50

In [101]: df1.join(df2, how='right') ❷
```

```
Out[101]:      A    B
          f  NaN  200
          b  200  150
          d  400   50

In [102]: df1.join(df2, how='inner') ❸
Out[102]:      A    B
          b  200  150
          d  400   50

In [103]: df1.join(df2, how='outer') ❹
Out[103]:      A    B
          a  100  NaN
          b  200  150
          c  300  NaN
          d  400   50
          f  NaN  200
```

❶ 左连接是默认操作。

❷ 右连接相当于颠倒了 DataFrame 对象的顺序。

❸ 内连接只保留在两个索引中都存在的索引值。

❹ 外连接保留两个索引中的所有索引值。

连接也可以基于空的 DataFrame 对象。在这种情况下，列按顺序被创建，这与左连接类似：

```
In [104]: df = pd.DataFrame()

In [105]: df['A'] = df1['A'] ❶

In [106]: df
Out[106]:      A
          a  100
          b  200
          c  300
          d  400

In [107]: df['B'] = df2 ❷

In [108]: df
Out[108]:      A    B
          a  100  NaN
          b  200  150
          c  300  NaN
          d  400   50
```

❶ `df1` 为第一列 A。

❷ `df2` 为第二列 B。

使用字典来组合数据集可以得到类似于外连接的结果，因为列是同时创建的：

```
In [109]: df = pd.DataFrame({'A': df1['A'], 'B': df2['B']}) ❶

In [110]: df
Out[110]:      A    B
          a  100  NaN
          b  200  150
          c  300  NaN
          d  400   50
          f  NaN  200
```

❶ `DataFrame` 对象的各列用作字典对象的值。

5.7.3 合并

连接操作根据待连接的 `DataFrame` 对象的索引进行,而合并操作通常在两个数据集共享的某列上进行。为此，在两个原始的 `DataFrame` 对象上添加一个新列 `C`：

```
In [111]: c = pd.Series([250, 150, 50], index=['b', 'd', 'c'])
          df1['C'] = c
          df2['C'] = c

In [112]: df1
Out[112]:      A      C
          a  100    NaN
          b  200  250.0
          c  300   50.0
          d  400  150.0

In [113]: df2
Out[113]:      B      C
          f  200    NaN
          b  150  250.0
          d   50  150.0
```

默认情况下，这种合并操作根据单一共享列 `C` 进行。但也存在其他可能，例如外合并：

```
In [114]: pd.merge(df1, df2) ❶
Out[114]:      A      C    B
          0  100    NaN  200
          1  200  250.0  150
```

```
          2  400  150.0   50

In [115]: pd.merge(df1, df2, on='C') ❶
Out[115]:      A      C    B
          0  100    NaN  200
          1  200  250.0  150
          2  400  150.0   50

In [116]: pd.merge(df1, df2, how='outer')❷
Out[116]:      A      C    B
          0  100    NaN  200
          1  200  250.0  150
          2  300   50.0  NaN
          3  400  150.0   50
```

❶ 默认合并根据 C 列进行。

❷ 也可以进行外合并，保留所有数据行。

可用的合并操作类型还有很多，下面的代码演示了其中的几种：

```
In [117]: pd.merge(df1, df2, left_on='A', right_on='B')
Out[117]:      A    C_x    B  C_y
          0  200  250.0  200  NaN

In [118]: pd.merge(df1, df2, left_on='A', right_on='B', how='outer')
Out[118]:      A    C_x    B    C_y
          0  100    NaN  NaN    NaN
          1  200  250.0  200    NaN
          2  300   50.0  NaN    NaN
          3  400  150.0  NaN    NaN
          4  NaN    NaN  150  250.0
          5  NaN    NaN   50  150.0

In [119]: pd.merge(df1, df2, left_index=True, right_index=True)
Out[119]:      A    C_x    B    C_y
          b  200  250.0  150  250.0
          d  400  150.0   50  150.0

In [120]: pd.merge(df1, df2, on='C', left_index=True)
Out[120]:      A      C    B
          f  100    NaN  200
          b  200  250.0  150
          d  400  150.0   50

In [121]: pd.merge(df1, df2, on='C', right_index=True)
```

```
Out[121]:      A      C    B
          a  100    NaN  200
          b  200  250.0  150
          d  400  150.0   50

In [122]: pd.merge(df1, df2, on='C', left_index=True, right_index=True)
Out[122]:      A    C      B
          b  200  250.0  150
          d  400  150.0   50
```

5.8　性能特征

本章的许多例子说明，用 pandas 实现同一个目标往往有多种选择。本节以按元素加总两列数值为例来这些选择进行对比。首先用 NumPy 生成数据集：

```
In [123]: data = np.random.standard_normal((1000000, 2)) ❶

In [124]: data.nbytes ❶
Out[124]: 16000000

In [125]: df = pd.DataFrame(data, columns=['x', 'y']) ❷

In [126]: df.info() ❷
          <class 'pandas.core.frame.DataFrame'>
          RangeIndex: 1000000 entries, 0 to 999999
          Data columns (total 2 columns):
          X    1000000 non-null float64
          Y    1000000 non-null float64
          dtypes: float64(2)
          memory usage: 15.3 MB
```

❶ 包含随机数的 `ndarray` 对象。

❷ 包含随机数的 `DataFrame` 对象。

其次，用于实现手边任务的一些选择的功能的性能还不错：

```
In [127]: %time res = df['x'] + df['y'] ❶
          CPU times: user 7.35 ms, sys: 7.43 ms, total: 14.8 ms
          Wall time: 7.48 ms

In [128]: res[:3]
Out[128]: 0    0.387242
          1   -0.969343
          2   -0.863159
```

```
         dtype: float64

In [129]: %time res = df.sum(axis=1) ❷
          CPU times: user 130 ms, sys: 30.6 ms, total: 161 ms
          Wall time: 101 ms

In [130]: res[:3]
Out[130]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64

In [131]: %time res = df.values.sum(axis=1) ❸
          CPU times: user 50.3 ms, sys: 2.75 ms, total: 53.1 ms
          Wall time: 27.9 ms

In [132]: res[:3]
Out[132]: array([ 0.3872424 , -0.96934273, -0.86315944])

In [133]: %time res = np.sum(df, axis=1) ❹
          CPU times: user 127 ms, sys: 15.1 ms, total: 142 ms
          Wall time: 73.7 ms

In [134]: res[:3]
Out[134]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64

In [135]: %time res = np.sum(df.values, axis=1) ❺
          CPU times: user 49.3 ms, sys: 2.36 ms, total: 51.7 ms
          Wall time: 26.9 ms

In [136]: res[:3]
Out[136]: array([ 0.3872424 , -0.96934273, -0.86315944])
```

❶ 直接使用列（`Series` 对象）是最快的方法。

❷ 调用 `DataFrame` 对象的 `sum` 方法来计算总和。

❸ 调用 `ndarray` 对象的 `sum` 方法来计算总和。

❹ 调用 `DataFrame` 对象的 `np.sum` 方法来计算总和。

❺ 调用 `ndarray` 对象的 `np.sum` 方法来计算总和。

最后，还有两种选择可以计算元素和，它们分别基于 `eval` 和 `apply` 方法[1]：

```
   The application of the eval() method requires the numexpr package to
be installed.
   In [137]: %time res = df.eval('x + y') ❶
             CPU times: user 25.5 ms, sys: 17.7 ms, total: 43.2 ms
             Wall time: 22.5 ms

   In [138]: res[:3]
   Out[138]: 0    0.387242
             1   -0.969343
             2   -0.863159
             dtype: float64

   In [139]: %time res = df.apply(lambda row: row['x'] + row['y'], axis=1) ❷
             CPU times: user 19.6 s, sys: 83.3 ms, total: 19.7 s
             Wall time: 19.9 s

   In [140]: res[:3]
   Out[140]: 0    0.387242
             1   -0.969343
             2   -0.863159
             dtype: float64
```

❶ `eval` 是专门用于计算（复杂）数值表达式的方法，可以直接处理数据列。

❷ 最慢的选择是逐行使用 `apply` 方法，这就像在 Python 级别上循环访问所有行。

明智的选择

pandas 常常为实现统一目标提供多个选择。如果不确定使用哪一个，那么在时间是关键因素的情况下比较各种选择，以选择性能最好的一种。在上面这个简单的例子里，不同选择的执行时间相差好几个数量级。

5.9 结语

pandas 是强大的数据分析工具，已经成为所谓的 *PyData* 栈的核心软件包。它的 `DataFrame` 类特别适合于处理任何类型的表格数据。这些对象上的大部分操作是向量化的，这不仅可以得到和 NumPy 类似的简洁代码，而且还可以得到高性能。此外，pandas

1　要想使用 `eval` 方法需要安装 numexpr 软件包。——原注

能够很方便地处理不完整数据集（NumPy 做不到这一点）。pandas 和 DataFrame 类将是本书后续多个章节的核心，在必要时将使用并介绍它们的更多特性。

5.10 延伸阅读

pandas 是一个开源项目，你既可以阅读在线文档，也可以下载其 PDF 版本[1]。官网上还提供了一些附加资源。

至于 NumPy，我们建议的 pandas 参考书如下。

- McKinney, Wes (2017). *Python for Data Analysis*. Sebastopol, CA: O'Reilly.
- VanderPlas, Jake (2016). *Python Data Science Handbook*. Sebastopol, CA: O'Reilly.

1 在本书编写期间，PDF 文档共有 2500 页。——原注

第 6 章

面向对象编程

软件工程的目的是控制复杂度，而不是制造复杂度。

——帕梅拉 · 扎维

面向对象编程（OOP）是当今非常流行的编程范型。这种方法若被正确使用，能够提供其他编程范型（如过程式编程）无法提供的众多优势。在许多情况下，OOP 好像特别适合金融建模和金融算法的实现。不过，这种方法也有许多批评者，他们声称对 OOP 的某个特征甚至整个范型表示怀疑。本章采取中立的态度，将 OOP 视为一种重要工具，它也许不是每个问题的最佳解决方案，但应该是金融业编程人员和宽客必备的工具之一。与 OOP 相伴的是一些新的术语，对本书和后续章节比较重要的如下所示。

类

　　某一类对象的抽象定义，例如人类。

对象

　　类的一个实例，例如 Sandra。

属性

　　类（类属性）或者类实例（实例属性）的特征。例如，“哺乳动物”“雌性”“雄性”或者“眼睛的颜色”。

方法

　　类可以实现的一个操作，例如步行。

参数

　　影响方法行为的输入，例如，三“步”。

实例化

　　根据某个抽象类创建具体对象的过程。

将这些概念转化成 Python 代码，实现“人类”的一个简单类示例的代码如下：

```
In [1]: class HumanBeing(object): ❶
            def __init__(self, first_name, eye_color): ❷
                self.first_name = first_name ❸
                self.eye_color = eye_color ❹
                self.position = 0 ❺
            def walk_steps(self, steps): ❻
                self.position += steps ❼
```

❶ 类定义语句，`self` 指类的当前实例。

❷ 实例化时调用的特殊方法。

❸ 用参数值初始化名字属性。

❹ 用参数值初始化眼睛颜色属性。

❺ 用 0 初始化位置属性。

❻ 步行方法定义，以 `steps`（步数）为参数。

❼ 按照 `steps` 值更改位置。

根据类定义，可以初始化并使用一个新的 Python 对象：

```
In [2]: Sandra = HumanBeing('Sandra', 'blue') ❶

In [3]: Sandra.first_name ❷
Out[3]: 'Sandra'

In [4]: Sandra.position ❷
Out[4]: 0

In [5]: Sandra.walk_steps(5) ❸

In [6]: Sandra.position ❹
Out[6]: 5
```

❶ 实例化。

❷ 访问属性值。

❸ 调用方法。

❹ 访问更新后的 `position` 值。

从人的因素方面考虑，OOP 有以下几个特征。

自然的思维方式

人类思维通常围绕现实世界或者抽象对象（如一辆汽车或者某种金融工具）发展。OOP 适合建立这些对象及其特性的模型。

降低复杂度

通过不同的方法，OOP 可以降低某个问题或者方法的复杂度，然后对其特征逐个建立模型。

更好的用户接口

在许多情况下，OOP 能实现更好的用户接口和更紧凑的代码。例如，观察 NumPy 的 `ndarray` 类或者 pandas 的 `DataFrame` 类就能明显地感受到这一点。

Python 风格的建模方式

不考虑 OOP 的优劣，它都是 Python 中占据主导地位的范型，也是“Python 中一切皆对象”这一说法的由来。OOP 还使编程人员可以构建自定义类，自定义类实例的表现和标准 Python 类的每个实例类似。

OOP 还有多种技术上的特征，具体如下。

抽象

使用属性和方法，我们可以构建抽象、灵活的对象模型，聚焦于有关的特征，忽略不需要的细节。在金融业务中，这可能意味着创建一个通用类，以抽象方式确立某种金融工具的模型。然后，该类的实例代表具体的金融产品，例如某个投资银行设计和提供的产品。

模块化

OOP 简化了将代码分解成多个模块、然后将其连接成完整的代码库的过程。例如，某股票的欧式期权可以通过一个或者两个类（一个用于目标股票，另一个用于期权本身）实现。

继承性

继承性指的是一个类可以从另一个类中继承属性和方法。在金融中，从通用金融工具开始，下一级可以是某类衍生品，然后是欧式期权，最后是欧式看涨期权。每个类可以从更高级别的类中继承属性和方法。

聚合

聚合指的是某个对象至少有一部分是由多个其他对象组成的情况，这些对象都可以独立存在。为欧式看涨期权建模的类可以包含其他对象的属性，这些对象分别

用于标的股票和对应的短期贴现率，它们也可以独立地供其他对象使用。

组合

组合类似于聚合，但单个对象不能独立于其他对象存在。考虑一个包含固定利率方和浮动利率方的定制利率掉期，提供不同利率的双方是不能独立于掉期存在的。

多态性

多态性可以呈现出多种形式。在 Python 环境中特别重要的是所谓的“鸭子类型”（Duck typing）。这个名词指的是标准操作可能在许多不同类上实现，它们的实例并不真正知道所处理对象的事实。对于一个金融工具类，这可能意味着，人们可以调用方法 `get_current_price`（获取当前价格），而无须考虑对象的具体类型（股票、期权、掉期）。

封装性

这个概念指的是类内部数据仅能通过公共方法访问。建立股票模型的类可能有这样一个属性 `current_stock_price`。封装性使这个属性只能通过 `get_current_stock_price()`方法访问，并对用户隐藏数据（也就是使其变成私有数据）。这种方法可以通过简单地处理（也可能更改）属性值来避免意料之外的影响。但是，在 Python 中指定私有数据的方法有一定限制。

从更高的层面讲，上述许多特征都可以概括为软件工程的两个总体目标，如下所示。

可重用性

继承性和多态性等概念改善了代码的可重用性，并提高了编程人员的效能和生产率，它们还简化了代码维护。

无冗余

与此同时，这些方法使程序员可构建几乎无冗余的代码，避免重复实现工作，减少调试、测试及维护所需的精力。它们还有可能得到更短小的代码库。

本章组织如下。

Python 对象简介

本节以 OOP 的视角来观察一些 Python 对象。

Python 类基础知识

本节介绍 Python OOP 的核心要素，使用金融工具和投资组合头寸作为主要示例。

Python 数据模型

本节讨论 Python 数据模型的重要元素，以及某些特殊方法所起的作用。

6.1 Python 对象简介

首先以 OOP 编程人员的视角，简单了解一下前几章介绍的一些标准对象。

6.1.1 int

从简单的整数（int）对象开始。即便是如此简单的 Python 对象，也有着重要的 OOP 特性：

```
In [7]: n = 5 ❶

In [8]: type(n) ❷
Out[8]: int

In [9]: n.numerator ❸
Out[9]: 5

In [10]: n.bit_length() ❹
Out[10]: 3

In [11]: n + n ❺
Out[11]: 10

In [12]: 2 * n ❻
Out[12]: 10

In [13]: n.__sizeof__() ❼
Out[13]: 28
```

❶ 新实例 n。

❷ 对象类型。

❸ 属性。

❹ 方法。

❺ 应用+运算符（加法）。

❻ 应用*运算符（乘法）。

❼ 调用特殊方法__sizeof__获得内存使用量[1]。

1 Python 中的特殊属性及方法用前导和后缀的两个下划线表示，例如在，__XYZ__().n.__sizeof__() 内部调用 import sys 中 sys.getsizeof(n)。——原注

6.1.2 list

列表（list）对象有更多的方法，但表现与整数对象基本相同：

```
In [14]: l = [1, 2, 3, 4] ❶

In [15]: type(l) ❷
Out[15]: list

In [16]: l[0] ❸
Out[16]: 1

In [17]: l.append(10) ❹

In [18]: l + l ❺
Out[18]: [1, 2, 3, 4, 10, 1, 2, 3, 4, 10]

In [19]: 2 * l ❻
Out[19]: [1, 2, 3, 4, 10, 1, 2, 3, 4, 10]

In [20]: sum(l) ❼
Out[20]: 20

In [21]: l.__sizeof__() ❽
Out[21]: 104
```

❶ 新实例 `l`。

❷ 对象类型。

❸ 通过索引选择一个元素。

❹ 方法。

❺ 应用+运算符（联接）。

❻ 应用*运算符（联接）。

❼ 应用标准 Python 函数 `sum`。

❽ 调用特殊方法`__sizeof__`获得内存使用量，以字节为单位。

6.1.3 ndarray

整数和列表对象是标准 Python 对象。NumPy 的 `ndarray` 对象是来自一个开源软件包的“定制”对象：

```
In [22]: import numpy as np ❶

In [23]: a = np.arange(16).reshape((4, 4)) ❷
```

```
In [24]: a ❷
Out[24]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])

In [25]: type(a) ❸
Out[25]: numpy.ndarray
```

❶ 导入 numpy。

❷ 新实例 a。

❸ 对象类型。

虽然 ndarray 对象不是标准对象，但它在许多情况下表现得就像标准对象——这要归功于 Python 数据模型，我们将在本章后面解释：

```
In [26]: a.nbytes ❶
Out[26]: 128

In [27]: a.sum() ❷
Out[27]: 120

In [28]: a.cumsum(axis=0) ❸
Out[28]: array([[ 0,  1,  2,  3],
                [ 4,  6,  8, 10],
                [12, 15, 18, 21],
                [24, 28, 32, 36]])

In [29]: a + a ❹
Out[29]: array([[ 0,  2,  4,  6],
                [ 8, 10, 12, 14],
                [16, 18, 20, 22],
                [24, 26, 28, 30]])

In [30]: 2 * a ❺
Out[30]: array([[ 0,  2,  4,  6],
                [ 8, 10, 12, 14],
                [16, 18, 20, 22],
                [24, 26, 28, 30]])
In [31]: sum(a) ❻
Out[31]: array([24, 28, 32, 36])

In [32]: np.sum(a) ❼
```

```
Out[32]: 120

In [33]: a.__sizeof__() ❽
Out[33]: 112
```

❶ 属性。

❷ 方法（聚合）。

❸ 方法（无聚合）。

❹ 应用+运算符（加法）。

❺ 应用*运算符（乘法）。

❻ 应用标准 Python 函数 `sum`。

❼ 应用 NumPy 通用函数 `np.sum`。

❽ 调用特殊方法`__sizeof__`，获得以字节表示的内存使用量。

6.1.4 DataFrame

我们最后来简单地观察一下 pandas 的 `DataFrame` 对象，其表现类似于 `ndarray` 对象。首先，以 `ndarray` 对象为基础实例化 `DataFrame` 对象：

```
In [34]: import pandas as pd ❶

In [35]: df = pd.DataFrame(a, columns=list('abcd')) ❷

In [36]: type(df) ❸
Out[36]: pandas.core.frame.DataFrame
```

❶ 导入 pandas。

❷ 新实例 `df`。

❸ 对象类型。

其次，观察一下对象的属性、方法和操作：

```
In [37]: df.columns ❶
Out[37]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [38]: df.sum() ❷
Out[38]: a    24
         b    28
         c    32
         d    36
         dtype: int64

In [39]: df.cumsum() ❸
```

```
Out[39]:     a   b  c  d
         0   0   1  2  3
         1   4   6  8 10
         2  12  15 18 21
         3  24  28 32 36

In [40]: df + df ❹
Out[40]:     a   b  c  d
         0   0   2  4  6
         1   8  10 12 14
         2  16  18 20 22
         3  24  26 28 30

In [41]: 2 * df ❺
Out[41]:     a   b  c  d
         0   0   2  4  6
         1   8  10 12 14
         2  16  18 20 22
         3  24  26 28 30

In [42]: np.sum(df) ❻
Out[42]: a    24
         b    28
         c    32
         d    36
         dtype: int64

In [43]: df.__sizeof__() ❼
Out[43]: 208
```

❶ 属性。

❷ 方法（聚合）。

❸ 方法（无聚合）。

❹ 应用+运算符（加法）。

❺ 应用*运算符（乘法）。

❻ 应用 NumPy 的通用函数 `np.sum`。

❼ 调用特殊方法`__sizeof__`，获得以字节表示的内存使用量。

6.2 Python 类基础知识

本节介绍在 Python 中使用 OOP 的重要概念和具体语法。我们现在要研究的是，构建自

定义类来为一些对象类型建模，这些类型的对象无法用现有的 Python 对象类型轻松、高效或正确地建模。我们自始至终都使用金融工具的例子来讲解。

两行代码就足以创建一个新的 Python 类：

```
In [44]: class FinancialInstrument(object): ❶
             Pass ❷

In [45]: fi = FinancialInstrument() ❸

In [46]: type(fi)❹
Out[46]: __main__.FinancialInstrument

In [47]: fi ❺
Out[47]: <__main__.FinancialInstrument at 0x116767278>

In [48]: fi.__str__() ❺
Out[48]: '<__main__.FinancialInstrument object at 0x116767278>'

In [49]: fi.price = 100 ❻

In [50]: fi.price ❻
Out[50]: 100
```

❶ 类定义语句[1]。

❷ 一些代码，这里简单地用 `pass` 关键字代替。

❸ 名为 `fi` 的新实例。

❹ 对象类型。

❺ 每个 Python 对象都有某些“特殊”的属性和方法（来自 `object`）；这里调用了读取字符串表示的特殊方法。

❻ 每个对象都可以在程序运行中定义所谓的数据属性——以常规属性对比。

`__init__`是一个重要的特殊方法，每当对象实例化时调用。它所取的参数为对象本身（惯例为 `self`）和可能的其他参数：

```
In [51]: class FinancialInstrument(object):
             author = 'Yves Hilpisch' ❶
             def __init__(self, symbol, price): ❷
                 self.symbol = symbol ❸
```

1 建议对类使用驼峰式命名法，在没有歧义的情况下，也可以使用小写或者蛇形命名法（如 `financial_instrument`）。——原注

```
                self.price = price ❸

In [52]: FinancialInstrument.author❶
Out[52]: 'Yves Hilpisch'

In [53]: aapl = FinancialInstrument('AAPL', 100) ❹

In [54]: aapl.symbol ❺
Out[54]: 'AAPL'

In [55]: aapl.author ❻
Out[55]: 'Yves Hilpisch'

In [56]: aapl.price = 105 ❼

In [57]: aapl.price ❼
Out[57]: 105
```

❶ 类属性定义（由每个实例继承）。

❷ 初始化时调用特殊方法`__init__`。

❸ 实例属性的定义（每个实例单独定义的）。

❹ 名为`fi`的新实例。

❺ 访问实例属性。

❻ 访问类属性。

❼ 更改实例属性值。

金融工具的价格经常变化，但是其代码可能不会改变。为了在类定义中引入封装，可以定义两个方法`get_price()`和`set_price()`。以下代码从前面的类定义（而不再是从`object`）继承：

```
In [58]: class FinancialInstrument(FinancialInstrument): ❶
             def get_price(self): ❷
                 return self.price ❷
             def set_price(self, price): ❸
                 self.price = price ❹

In [59]: fi = FinancialInstrument('AAPL', 100) ❺

In [60]: fi.get_price() ❻
Out[60]: 100

In [61]: fi.set_price(105) ❼
```

```
In [62]: fi.get_price() ❻
Out[62]: 105

In [63]: fi.price ❽
Out[63]: 105
```

❶ 通过前一版本继承的类定义。

❷ 定义 `get_price()`方法。

❸ 定义 `set_price()`方法。

❹ 根据参数值更新实例属性。

❺ 基于新类定义的新实例 `fi`。

❻ 调用 `get_price()`方法读取实例属性值。

❼ 通过 `set_price()`更新实例属性值。

❽ 直接访问实例属性。

封装的目标通常是对使用类的用户隐藏数据。添加取值（*getter*）和设值（*setter*）方法是实现这一目标的一部分。不过，这并不能阻止用户直接访问和操纵实例属性。此时，私有实例属性就有了用武之地。它们以两个前导下划线定义：

```
In [64]: class FinancialInstrument(object):
             def __init__(self, symbol, price):
                 self.symbol = symbol
                 self.__price = price ❶
             def get_price(self):
                 return self.__price
             def set_price(self, price):
                 self.__price = price

In [65]: fi = FinancialInstrument('AAPL', 100)

In [66]: fi.get_price() ❷
Out[66]: 100

In [67]: fi.__price ❸

         ----------------------------------------------------------------
         AttributeError                    Traceback (most recent call last)
         <ipython-input-67-bd62f6cadb79> in <module>
         ----> 1 fi.__price ❸

         AttributeError: 'FinancialInstrument' object has no attribute '__price'
```

```
In [68]: fi._FinancialInstrument__price ❹
Out[68]: 100

In [69]: fi._FinancialInstrument__price = 105 ❹
In [70]: fi.set_price(100) ❺
```

❶ 将价格定义为私有实例属性。

❷ `get_price()`返回该属性值。

❸ 试图直接访问属性导致出错。

❹ 如果在类名前加上一个前导下划线，那么直接访问和操纵仍然是有可能的。

❺ 将价格设回原始值。

> **Python 中的封装**
>
> 虽然 Python 类的封装基本上可以通过实例属性和对应的处理方法实现，但它并不能完全对用户隐藏数据。从这个意义上说，该特性更多的是 Python 的设计原则，而非 Python 类的技术特征。

考虑为某金融工具投资组合进行建模的另一个类。聚合的概念很容易用两个类来说明。`PortfolioPosition` 类以 `FinancialInstrument` 类的一个实例作为属性值。我们可以添加一个属性值 `position_size` 来计算头寸价值：

```
In [71]: class PortfolioPosition(object):
             def __init__(self, financial_instrument, position_size):
                 self.position = financial_instrument ❶
                 self.__position_size = position_size ❷
             def get_position_size(self):
                 return self.__position_size
             def update_position_size(self, position_size):
                 self.__position_size = position_size
             def get_position_value(self):
                 return self.__position_size * \
                        self.position.get_price() ❸

In [72]: pp = PortfolioPosition(fi, 10)

In [73]: pp.get_position_size()
Out[73]: 10

In [74]: pp.get_position_value() ❸
Out[74]: 1000

In [75]: pp.position.get_price() ❹
```

```
Out[75]: 100

In [76]: pp.position.set_price(105) ❺

In [77]: pp.get_position_value() ❻
Out[77]: 1050
```

❶ 基于 `FinancialInstrument` 类实例的一个实例属性。

❷ `PortfolioPosition` 类的私有实例属性。

❸ 根据属性计算头寸价值。

❹ 连接到实例属性对象的方法可以直接访问（也可以隐藏）。

❺ 更新金融工具价格。

❻ 根据更新后的价格以计算新的头寸价值。

6.3 Python 数据模型

6.2 节的例子展示了所谓 Python 数据或对象模型的一些特征。Python 数据模型使你可以设计一直与 Python 基本语言结构进行交互的类。这些类支持以下任务和结构：

- 迭代；
- 集合处理；
- 属性访问；
- 运算符重载；
- 函数与方法调用；
- 对象创建与销毁；
- 字符串表示（例如，用于打印）。
- 托管上下文（即 with 块）。

由于 Python 数据模型非常重要，所以本节将专门用一个例子来探索它的多个特征。这个例子实现了一个类，该类用于 3 个元素的一维向量（想象欧几里得空间里的向量）。首先是特殊方法`__init__`：

```
In [78]: class Vector(object):
             def __init__(self, x=0, y=0, z=0): ❶
                 self.x = x ❶
                 self.y = y ❶
```

```
                self.z = z ❶

In [79]: v = Vector(1, 2, 3) ❷

In [80]: v ❸
Out[80]: <__main__.Vector at 0x1167789e8>
```

❶ 3 个预先初始化的实例属性（想象三维空间）。

❷ 名为 v 的类新实例。

❸ 默认字符串表示。

特殊方法__repr__可以自定义字符串表示：

```
In [81]: class Vector(Vector):
             def __repr__(self):
                 return 'Vector(%r, %r, %r)' % (self.x, self.y, self.z)

In [82]: v = Vector(1, 2, 3)

In [83]: v ❶
Out[83]: Vector(1, 2, 3)

In [84]: print(v) ❶
         Vector(1, 2, 3)
```

❶ 新的字符串表示。

abs()和 bool()是两个 Python 的标准函数，它们在 Vector 类上的表现可以通过特殊方法__abs__和__bool__定义：

```
In [85]: class Vector(Vector):
             def __abs__(self):
                 return (self.x ** 2 + self.y ** 2 +
                         self.z ** 2) ** 0.5❶

             def __bool__(self):
                 return bool(abs(self))

In [86]: v = Vector(1, 2, -1) ❷

In [87]: abs(v)
Out[87]: 2.449489742783178

In [88]: bool(v)
Out[88]: True
```

```
In [89]: v = Vector() ❸

In [90]: v ❸
Out[90]: Vector(0, 0, 0)

In [91]: abs(v)
Out[91]: 0.0

In [92]: bool(v)
Out[92]: False
```

❶ 根据 3 个属性值返回欧几里得范数。

❷ 具有非零属性值的新 `Vector` 对象。

❸ 仅包含零值的新 `Vector` 对象。

前面已经多次看到，`+`和`*`运算符可应用于几乎任何 Python 对象。其行为通过特殊方法`__add__`和`__mul__`定义：

```
In [93]: class Vector(Vector):
             def __add__(self, other):
                 x = self.x + other.x
                 y = self.y + other.y
                 z = self.z + other.z
                 return Vector(x, y, z)❶

             def __mul__(self, scalar):
                 return Vector(self.x * scalar,
                               self.y * scalar,
                               self.z * scalar)❶

In [94]: v = Vector(1, 2, 3)

In [95]: v + Vector(2, 3, 4)
Out[95]: Vector(3, 5, 7)

In [96]: v * 2
Out[96]: Vector(2, 4, 6)
```

❶ 在这个例子中，每个特殊方法返回一个属于自身类型的对象。

`Len()`是另一个 Python 标准函数，它返回以元素数量表示的对象长度。在对象上调用这个函数时，就会访问特殊方法`__len__`。另一方面，特殊方法`__getitem__`通过方括号来实现索引：

```
In [97]: class Vector(Vector):
             def __len__(self):
                 return 3 ❶
             def __getitem__(self, i):
                 if i in [0, -3]: return self.x
                 elif i in [1, -2]: return self.y
                 elif i in [2, -1]: return self.z
                 else: raise IndexError('Index out of range.')

In [98]: v = Vector(1, 2, 3)

In [99]: len(v)
Out[99]: 3

In [100]: v[0]
Out[100]: 1

In [101]: v[-2]
Out[101]: 2

In [102]: v[3]

          ------------------------------------------------------------
          IndexError                  Traceback (most recent call last)
          <ipython-input-102-f998c57dcc1e> in <module>
          ----> 1 v[3]

          <ipython-input-97-b0ca25eef7b3> in __getitem__(self, i)
                7         elif i in [1, -2]: return self.y
                8         elif i in [2, -1]: return self.z
          ----> 9         else: raise IndexError('Index out of range.')

          IndexError: Index out of range.
```

❶ `Vector` 类的所有实例长度均为 3。

最后，特殊方法`__iter__`定义对象的元素在迭代时的行为，定义该操作的对象称为可迭代对象。所有集合和容器都是可迭代的：

```
In [103]: class Vector(Vector):
              def __iter__(self):
                  for i in range(len(self)):
                      yield self[i]

In [104]: v = Vector(1, 2, 3)
```

```
In [105]: for i in range(3): ❶
              print(v[i]) ❶
          1
          2
          3
In [106]: for coordinate in v: ❷
              print(coordinate) ❷
          1
          2
          3
```

❶ 用索引值间接迭代（通过`__getitem__`）。

❷ 在类实例上直接迭代（用`__iter__`）。

> **增强 Python**
> Python 数据模型可以定义与标准 Python 运算符、函数等无缝交互的类。这使 Python 成为一种相当灵活的编程语言，很容易通过新类和对象类型增强。

6.4 节将在单个代码块中提供一个 `Vector` 类定义来作为总结。

6.4 Vector 类

Vector 类代码如下：

```
In [107]: class Vector(object):
              def __init__(self, x=0, y=0, z=0):
                  self.x = x
                  self.y = y
                  self.z = z

              def __repr__(self):
                  return 'Vector(%r, %r, %r)' % (self.x, self.y, self.z)

              def __abs__(self):
                  return (self.x ** 2 + self.y ** 2 + self.z ** 2) ** 0.5

              def __bool__(self):
                  return bool(abs(self))
              def __add__(self, other):
                  x = self.x + other.x
                  y = self.y + other.y
```

```
        z = self.z + other.z
        return Vector(x, y, z)
    def __mul__(self, scalar):
        return Vector(self.x * scalar,
                      self.y * scalar,
                      self.z * scalar)

    def __len__(self):
        return 3

    def __getitem__(self, i):
        if i in [0, -3]: return self.x
        elif i in [1, -2]: return self.y
        elif i in [2, -1]: return self.z
        else: raise IndexError('Index out of range.')

    def __iter__(self):
        for i in range(len(self)):
            yield self[i]
```

6.5　结语

本章从理论和 Python 示例，介绍了面向对象编程的概念及方法。OOP 是 Python 使用的主要编程范型之一，它不仅可以建模和实现相当复杂的应用，而且由于其灵活的 Python 数据模型，还可以创建与标准 Python 对象表现相似的自定义对象。虽然针对 OOP 的批评很多，但可以肯定地说，它为 Python 编程人员和宽客（quent）提供了强大的工具，这些工具在程序达到一定复杂度时很有帮助。第 5 部分开发和讨论的衍生品定价软件包提供了一个案例，其中 OOP 似乎是解决固有复杂性和抽象要求的唯一合理编程范型。

6.6　延伸阅读

下面是有关 OOP、特别是 Python 编程与 OOP 结合的优质文章：

- *Lecture Notes on Object-Oriented Programming*；
- *Object-Oriented Programming in Python*。

下面是关于 Python 面向对象编程及 Python 数据模型的一本出色图书：

- Ramalho, Luciano (2016). *Fluent Python*. Sebastopol, CA: O’Reilly.

第 3 部分

金融数据科学

本书的第 3 部分介绍金融数据科学的相关基本技术、方法和软件包。本部分的许多主题（如可视化）和软件包（如 scikit-learn）是 Python 数据科学的基础。从这个意义上说，这一部分为宽客及金融分析人员提供了成为金融数据科学家所需的 Python 工具。

和第 2 部分一样，本部分的章节按照主题组织，可以作为读者阅读的参考：

- 第 7 章讨论使用 `matplotlib` 和 `plotly` 进行的静态和交互式可视化；
- 第 8 章介绍用 `pandas` 处理金融时间序列数据的方法；
- 第 9 章的焦点是正确、快速地输入/输出（I/O）操作；
- 第 10 章介绍加快 Python 代码运行速度的方法；
- 第 11 章聚焦于金融中经常需要的数学工具；
- 第 12 章介绍如何使用 Python 实现推断统计学方法；
- 第 13 章介绍统计学与机器学习方法。

第 7 章

数据可视化

一图胜千言。

——阿瑟 · 布里斯班

本章介绍的是 matplotlib 和 plotly 的基本可视化功能。尽管有许多其他的可视化库，但是 matplotlib 已经确立了一个标杆，在许多情况下，它都是健壮、可靠的可视化工具。对于标准的绘图工作，它很容易理解，进行更复杂的绘图和自定义时，它又很灵活。此外，它可以与 NumPy、pandas 及其提供的数据结构紧密集成。

matplotlib 只能生成位图（如 PNG 或者 JPG 格式）形式的图表。相反，现代 Web 技术［如基于数据驱动文档（D3.js）标准的技术］可以生成很好的交互式可嵌入图表（如可以放大以检查某部分细节的交互式图表）。plotly 是一个创建 D3.js 图表的便利程序库。另一个较小的程序库 Cufflinks 能将 plotly 与 pandas 的 `DataFrame` 对象紧密集成，从而创建流行的金融图表（例如蜡烛图）。

本章主要介绍以下内容。

静态 2D 绘图

本节介绍 matplotlib，并介绍一组典型的 2D 图表。该 2D 图表包含从最简单的图形到具有两种刻度或者不同子图的较高级的图形。

静态 3D 绘图

本节以 matplotlib 为基础，介绍与某些金融应用有关的一组 3D 图表。

交互式 2D 绘图

本节介绍创建交互式 2D 图表所用的 plotly 和 Cufflinks。利用 Cufflinks 的 `QuantFigure` 功能，本节还会介绍典型的金融图表，例如用于股票技术分析的图表。

本章无法全面地介绍 Python、matplotlib 和 plotly 所能进行的数据可视化，但是提供了对金融学最基本和最重要的功能示例。后续的章节中还可以看到其他示例。例如，第 8 章更深入地说明了如何用 pandas 库可视化金融时间序列数据。

7.1　静态 2D 绘图

在创建样板数据并开始绘图之前，需要进行一些导入和定制工作：

```
In [1]: import matplotlib as mpl ❶

In [2]: mpl.__version__ ❷
Out[2]: '3.0.0'

In [3]: import matplotlib.pyplot as plt ❸

In [4]: plt.style.use('seaborn') ❹

In [5]: mpl.rcParams['font.family'] = 'serif' ❺

In [6]: %matplotlib inline
```

❶ 导入 `matplotlib`，平常使用缩写 `mpl`。

❷ 使用的 `matplotlib` 版本。

❸ 导入主绘图（子）库，平常缩写为 `plt`。

❹ 将绘图样式设置为 `seaborn`。

❺ 将所有图表的字体设置为 `serif`。

7.1.1　一维数据集

pyplot 子库中的 plot 函数是最基础的绘图函数，它相当强大。原则上，它需要两组数值。

***x* 值**

包含 *x* 坐标（横坐标）的列表或者数组。

***y* 值**

包含 *y* 坐标（纵坐标）的列表或者数组。

当然，*x* 和 *y* 值的数量必须相等，考虑如下代码，其输出如图 7-1 所示。

```
In [7]: import numpy as np

In [8]: np.random.seed(1000) ❶

In [9]: y = np.random.standard_normal(20) ❷

In [10]: x = np.arange(len(y)) ❸
         plt.plot(x, y); ❹
```

❶ 设置固定的随机数生成器种子，以便重现。

❷ 提取随机数（*y* 值）。

❸ 设置固定的整数值（*x* 值）。

❹ 以 *x* 和 *y* 对象调用 `plt.plot()`函数。

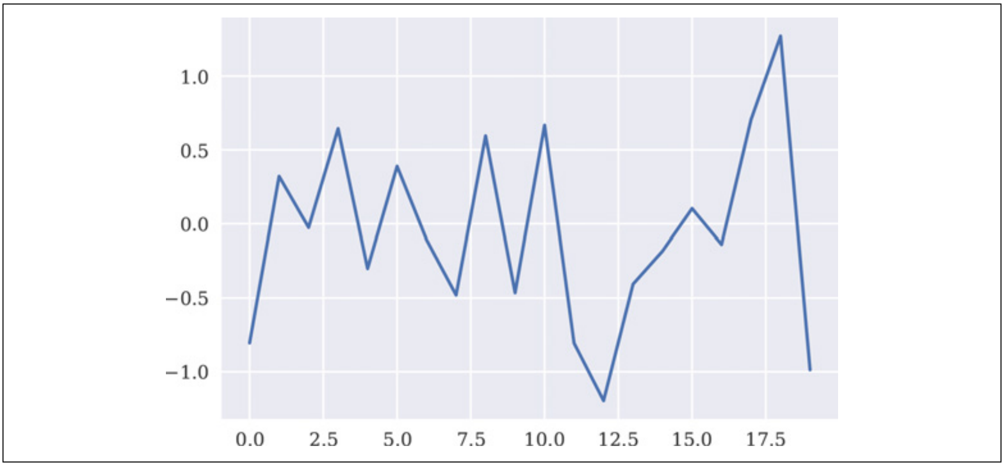


图 7-1　根据给定的 *x* 和 *y* 值绘图

`plt.plot()`会注意到何时传递了 `ndarray` 对象。在这种情况下，没有必要提供 *x* 值的“额外”信息。如果只提供 *y* 值，`plot` 以索引值作为对应的 *x* 值。因此，下面一行代码会生成完全一样的输出（结果见图 7-2）：

```
In [11]: plt.plot(y);
```



NumPy **数组和** matplotlib

可以简单地向 `matplotlib` 函数传递 NumPy 的 `ndarray` 对象。`matplotlib` 能够解释数据结构以简化绘图工作。但是要注意，不要传递太大或者太复杂的数组。

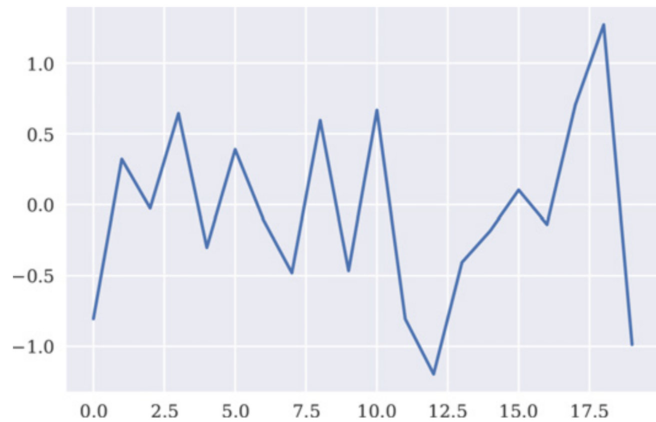


图 7-2　按照 ndarray 对象给出的数据绘图

由于大部分 `ndarray` 方法返回的仍然是 `ndarray` 对象，所以也可以附加一个方法（有些时候甚至可以附加多个方法）来传递对象。用样板数据调用 `ndarray` 对象上的 `cumsum()` 方法来获得这些数据的总和。和预想的一样，我们会得到不同的输出（见图 7-3）。

```
In [12]: plt.plot(y.cumsum());
```

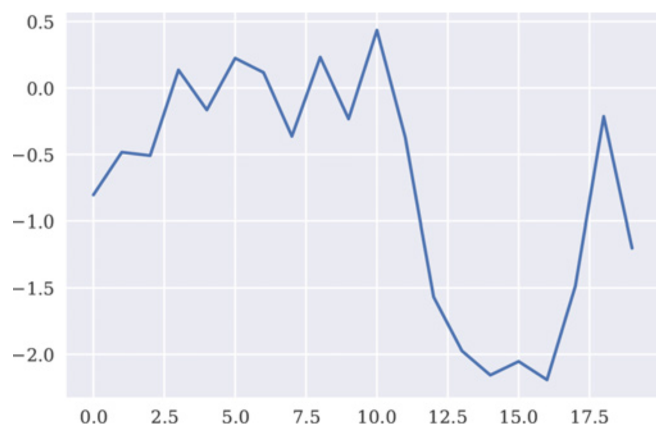


图 7-3　按照给定的 ndarray 对象和一个附加方法绘图

一般来说，默认绘图样式不能满足报表、出版等典型要求。例如，你可能希望自定义所使用的字体（例如，为了 LaTeX 字体兼容），在坐标轴上有标签或者为了更好的可辨认性而绘制网格。因此，matplotlib 提供了大量的函数以自定义绘图样式。有些函数容易理解，其他的则需要进行深入一步的研究。例如，操纵坐标轴和增加网格及标签的函数很容易理解（参见图 7-4）：

```
In [13]: plt.plot(y.cumsum())
         plt.grid(False) ❶
         plt.axis('equal'); ❷
```

❶ 关闭网格。

❷ 两个轴使用相同的刻度。

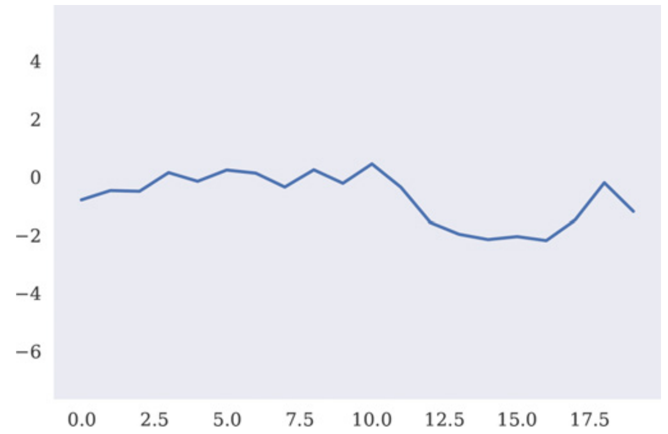


图 7-4　无网格图表

表 7-1 列出了 `plt.axis` 的其他选项，大部分都以字符串对象的形式传递。

表 7-1　**`plt.axis` 选项**

参数	描述
`Empty`	返回当前坐标轴限值
`off`	关闭坐标轴线和标签
`equal`	使用等刻度
`scaled`	通过尺寸变化平衡刻度
`tight`	使所有数据可见（缩小限值）
`image`	使所有数据可见（使用数据限值）
`[xmin, xmax, ymin, ymax]`	将设置限制为给定的（一组）值

此外，可以使用 `plt.xlim()` 和 `plt.ylim()` 设置每个坐标轴的最小值和最大值。下面的代码提供了一个示例，输出如图 7-5 所示。

```
In [14]: plt.plot(y.cumsum())
         plt.xlim(-1, 20)
         plt.ylim(np.min(y.cumsum()) - 1,
                  np.max(y.cumsum()) + 1);
```

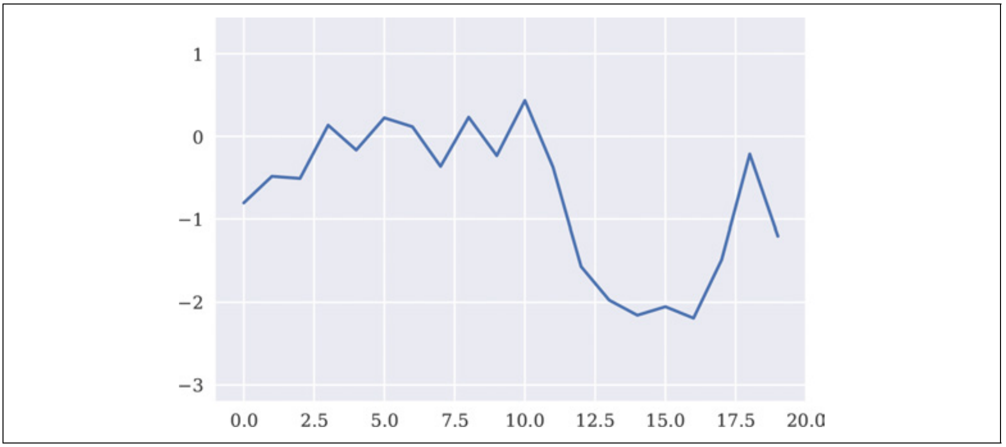


图 7-5　使用自定义坐标轴限值绘制图表

为了更容易让读者理解，图表通常包含一些标签——描述 *x* 和 *y* 值性质的标题和标签。这些标签分别通过 `plt.title()`、`plt.xlabel()` 和 `plt.ylabel()` 添加。默认情况下，即使提供的数据点是离散的，`plot()` 也绘制连续线条。离散点的绘制通过选择不同的样式选项实现。图 7-6 显示了点和线宽为 1.5 个点的线条。

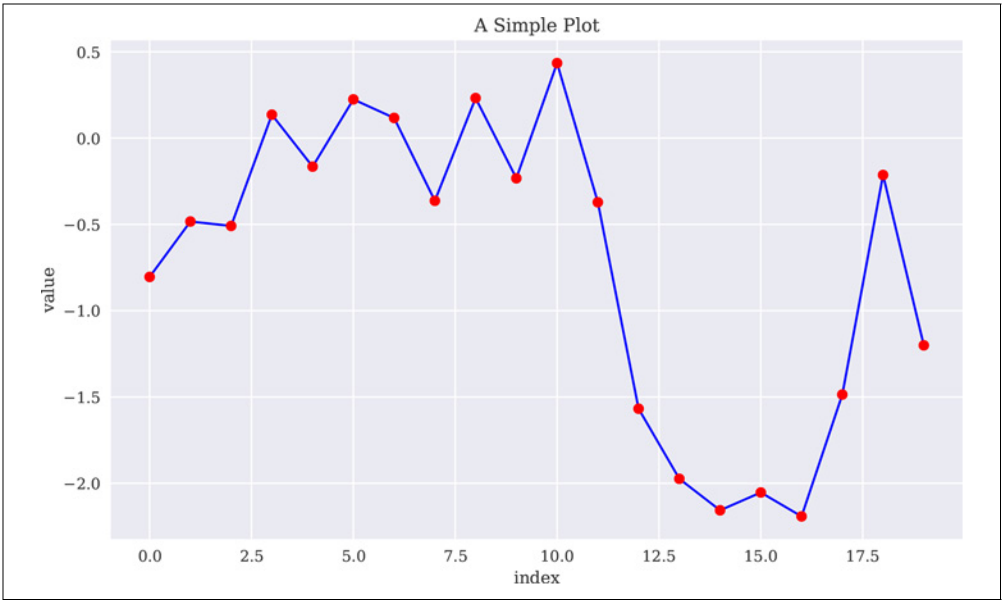


图 7-6　带有典型标签的图表

```
In [15]: plt.figure(figsize=(10, 6)) ❶
         plt.plot(y.cumsum(), 'b', lw=1.5) ❷
```

```
plt.plot(y.cumsum(), 'ro') ❸
plt.xlabel('index') ❹
plt.ylabel('value') ❺
plt.title('A Simple Plot'); ❻
```

❶ 增大图表的尺寸。

❷ 按照数据绘制一条线宽为 1.5 个点的蓝色线。

❸ 按照数据绘制粗（红色）点。

❹ 在 *x* 轴上加标签。

❺ 在 *y* 轴上加标签。

❻ 加一个标题。

默认情况下，`plt.plot()`支持表 7-2 中的颜色缩写。

表 7-2　标准颜色缩写

字符	颜色
b	蓝
g	绿
r	红
c	青
m	品红
y	黄
k	黑
w	白

至于线和点的样式，`plt.plot()`支持表 7-3 中列出的字符。

表 7-3　标准样式字符

字符	象征
-	实线样式
--	短画线样式
-.	点实线样式
:	虚线样式
.	点标记
,	像素标记
o	圆标记

续表

字符	象征
V	向下三角形标记
0	向上三角形标记
<	向左三角形标记
>	向右三角形标记
1	Tri_down 标记
2	Tri_up 标记
3	Tri_left 标记
4	Tri_right 标记
s	方形标记
p	五边形标记
0	星号
h	六角形标记 1
H	六角形标记 2
0	加号
x	X 标记
D	菱形标记
d	细菱形标记
|	垂直线标记
_	水平线标记

颜色缩写可以与样式字符任意组合，这样，可以确保用户能够轻松区分不同的数据集。绘图样式也会反映到图例中。

7.1.2　二维数据集

绘制一维数据可以看作一种特例。一般来说，数据集包含多个单独的子集。这种数据的处理遵循 matplotlib 处理一维数据时的原则。但是，这种情况会引起其他一些问题，例如，两个数据集可能有不同的刻度，无法用相同的 *y* 或 *x* 轴刻度绘制。另一个问题是，你可能希望以不同的方式可视化两组不同数据，例如，一组数据使用线图，另一组使用柱状图。

下面的代码生成了一个二维样本数据集——包含标准正态分布（伪）随机数的 20×2 `ndarray` 对象。在这个数组上调用 `cumsum()`来计算样本数据在 0 轴（即第一维）上的总和：

```
In [16]: y = np.random.standard_normal((20, 2)).cumsum(axis=0)
```

一般来说，也可以将这样的二维数组传递给 plt.plot()。它将自动把包含的数据解释为单独的数据集（沿着 1 轴，即第二维）。生成的图表如图 7-7 所示。

```
In [17]: plt.figure(figsize=(10, 6))
         plt.plot(y, lw=1.5)
         plt.plot(y, 'ro')
         plt.xlabel('index')
         plt.ylabel('value')
         plt.title('A Simple Plot');
```

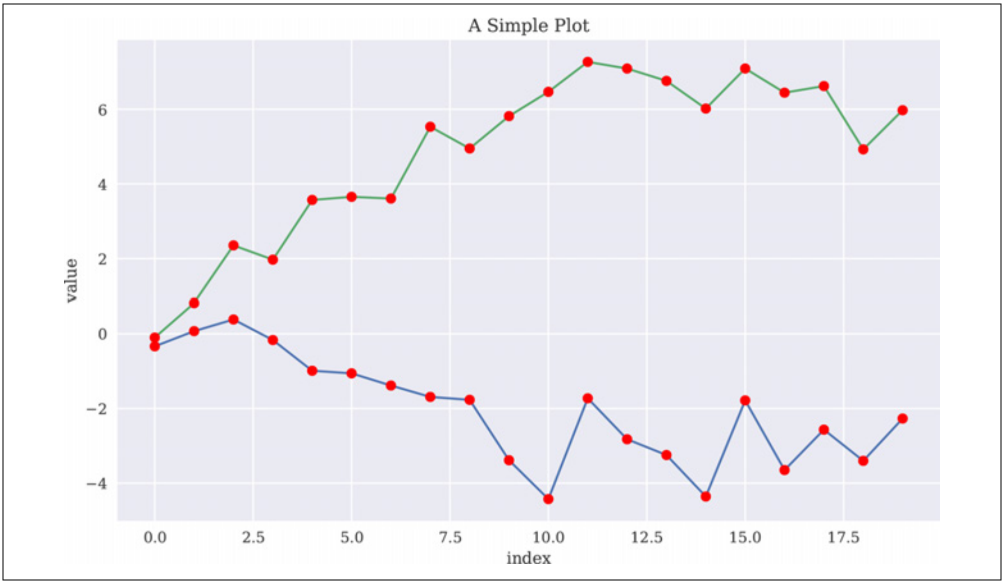


图 7-7　按照两个数据集绘图

在这种情况下，进一步的注释有助于用户更好地理解图表。你可以为每个数据集添加单独的标签并在图例中列出。plt.legend()接受不同的位置参数。0 表示“最佳位置”，也就是图例应尽可能少地遮盖数据。

图 7-8 展示了包含两个数据集的图表，这一次带有图例。在生成代码中，我们没有传递整个 ndarray 对象，而是分别访问两个数据子集（y[:, 0]和 y[:, 1]），这允许你为它们附加单独的标签：

```
In [18]: plt.figure(figsize=(10, 6))
         plt.plot(y[:, 0], lw=1.5, label='1st') ❶
         plt.plot(y[:, 1], lw=1.5, label='2nd') ❶
         plt.plot(y, 'ro')
         plt.legend(loc=0) ❷
         plt.xlabel('index')
         plt.ylabel('value')
```

```
            plt.title('A Simple Plot');
```

❶ 为数据子集定义标签。

❷ 将图例放在“最佳”位置。

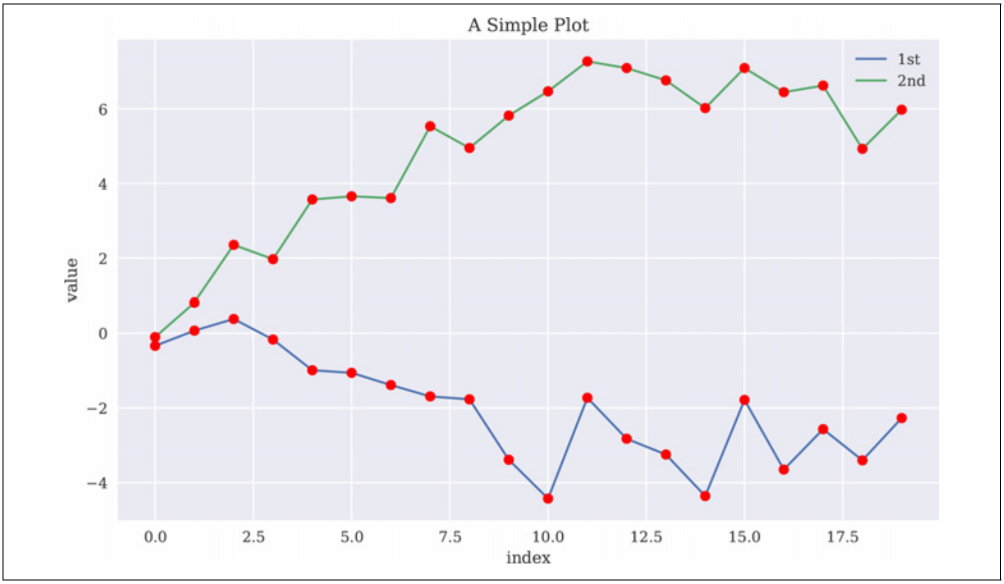


图 7-8　带有数据集标签的图表

`plt.legend()`的其他位置选项如表 7-4 所示。

表 7-4　plt.legend()选项

位置代码	描述
`Default`（默认）	右上
0	最佳
1	右上
2	左上
3	左下
4	右下
5	右侧
6	左侧居中
7	右侧居中
8	下方居中
9	上方居中
10	居中

多个具有类似刻度的数据集（如同一金融风险因素的模拟路径）可以用相同的 y 轴绘制。但是，数据集常常有不同的刻度，用单一 y 轴刻度绘制这种数据的图表通常会导致可视化信息的显著丢失。为了说明这种效果，我们将两个数据子集中的第一个数据集扩大 100 倍，然后再次绘制该图（参见图 7-9）。

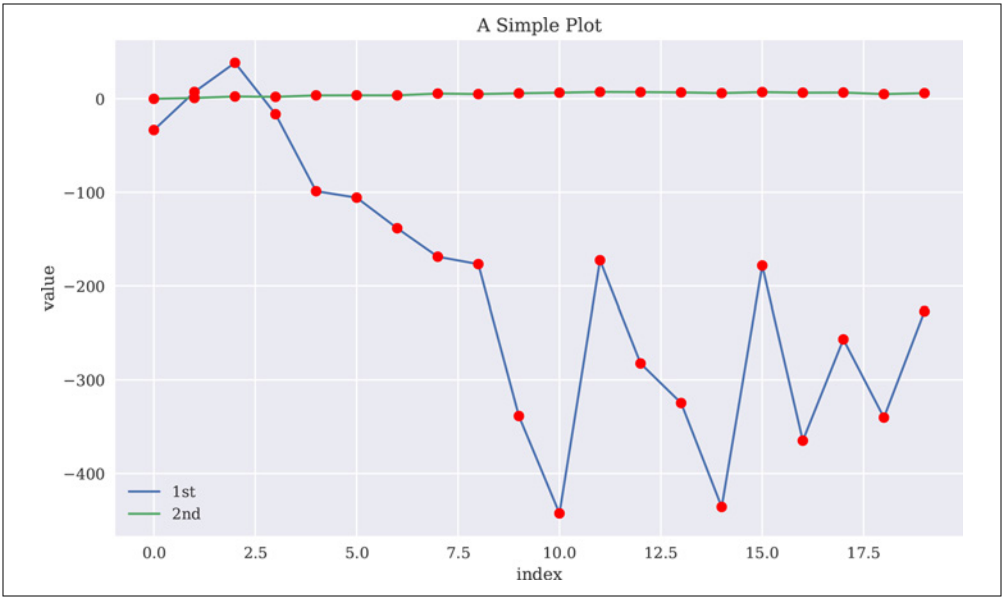


图 7-9　包含两个不同刻度数据集的图表

```
In [19]: y[:, 0] = y[:, 0] * 100 ❶

In [20]: plt.figure(figsize=(10, 6))
         plt.plot(y[:, 0], lw=1.5, label='1st')
         plt.plot(y[:, 1], lw=1.5, label='2nd')
         plt.plot(y, 'ro')
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.ylabel('value')
         plt.title('A Simple Plot');
```

❶ 改变第一个数据子集的刻度。

观察图 7-9 可以看出，第一个数据集仍然是“在视觉上易于辨认的”，而第二个数据集在新的 y 轴刻度上看起来像一条直线。在某种程度上，第二个数据集的有关信息现在“在视觉上已经丢失”。通过绘图手段而非调整数据（例如通过改变刻度）来解决这个问题的基本方法有两种：

- 使用 2 个 y 轴（左/右）；

- 使用两个子图（上/下，左/右）。

下面的例子在图表中引入第二个 *y* 轴。图 7-10 中有两个不同的 *y* 轴，左侧的 *y* 轴用于第一个数据集，右侧的 *y* 轴用于第二个数据集，因此，有两个图例：

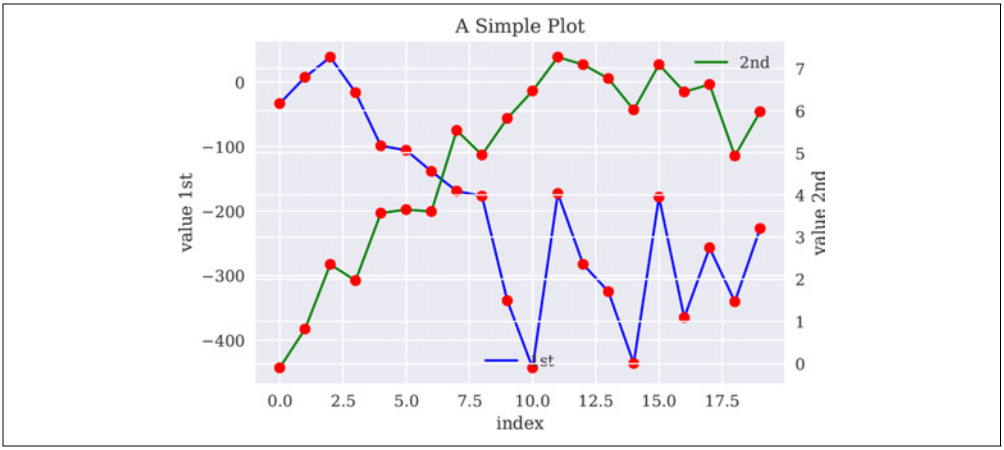


图 7-10　包含两个数据集、两个 *y* 轴的图表

```
In [21]: fig, ax1 = plt.subplots() ❶
         plt.plot(y[:, 0], 'b', lw=1.5, label='1st')
         plt.plot(y[:, 0], 'ro')
         plt.legend(loc=8)
         plt.xlabel('index')
         plt.ylabel('value 1st')
         plt.title('A Simple Plot')
         ax2 = ax1.twinx() ❷
         plt.plot(y[:, 1], 'g', lw=1.5, label='2nd')
         plt.plot(y[:, 1], 'ro')
         plt.legend(loc=0)
         plt.ylabel('value 2nd');
```

❶ 定义图与轴对象。

❷ 创建第二个轴对象，共享 *x* 轴。

管理坐标轴的代码行是关键，如：

```
fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
```

通过使用 `plt.subplots()` 函数，我们可以直接访问底层绘图对象（图、子图等）。例如，可以用它生成和第一个子图共享 *x* 轴的第二个子图。图 7-10 中有两个相互重叠的子图。

接下来，考虑两个单独子图的情况。这个选项提供了处理两个数据集的更大自由度，

如图 7-11 所示。

```
In [22]: plt.figure(figsize=(10, 6))
         plt.subplot(211) ❶
         plt.plot(y[:, 0], lw=1.5, label='1st')
         plt.plot(y[:, 0], 'ro')
         plt.legend(loc=0)
         plt.ylabel('value')
         plt.title('A Simple Plot')
         plt.subplot(212) ❷
         plt.plot(y[:, 1], 'g', lw=1.5, label='2nd')
         plt.plot(y[:, 1], 'ro')
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.ylabel('value');
```

❶ 定义上方的子图 1。

❷ 定义下方的子图 2。

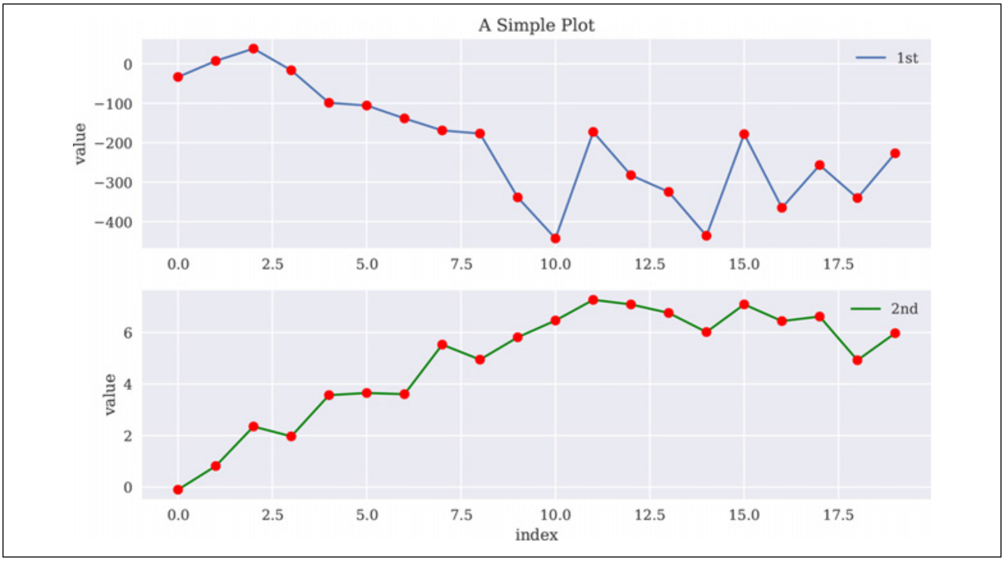


图 7-11　带有两个子图的图表

matplotlib 图形对象中子图的定位通过一种特殊的坐标系来实现。plt.subplot() 有 3 个整数参数，即 numrows、numcols 和 fignum（可能有逗号分隔，也可能没有）。numrows 指定行数，numcols 指定列数，fignum 指定子图编号（从 1 到 numrows × numcols）。例如，有 9 个大小相同子图的图表的表示应该为 numrows=3，numcols=3，fignum=1，2，…，9。左下角的子图“坐标”如下：

```
plt.subplot(3, 3, 9).
```

有时候，选择两个不同的图表类型来可视化数据可能是必要的，也是理想的。利用子图方法，我们就可以自由地组合 matplotlib 提供的任意图表类型[1]了。

图 7-12 组合了线图/点图和柱状图：

```
In [23]: plt.figure(figsize=(10, 6))
         plt.subplot(121)
         plt.plot(y[:, 0], lw=1.5, label='1st')
         plt.plot(y[:, 0], 'ro')
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.ylabel('value')
         plt.title('1st Data Set')
         plt.subplot(122)
         plt.bar(np.arange(len(y)), y[:, 1], width=0.5,
                 color='g', label='2nd') ❶
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.title('2nd Data Set');
```

❶ 创建一个柱状（bar）子图。

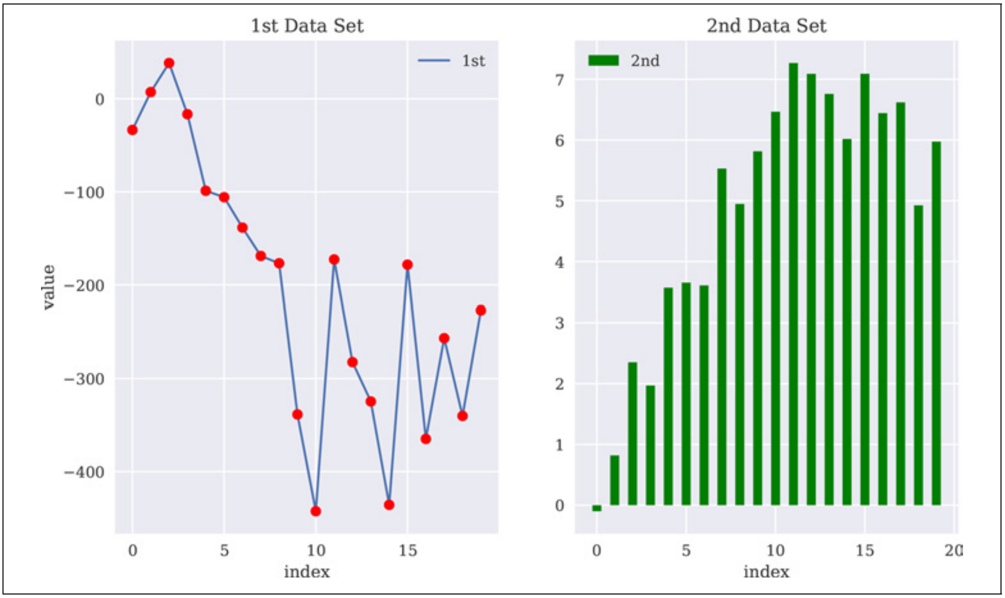


图 7-12　组合线/点子图和柱状子图

1　可用图表类型的概述可访问 matplotlib。——原注

7.1.3　其他绘图样式

对于二维绘图，线图和点图可能是金融学中最重要的；这是因为许多数据集用于表示时间序列数据，此类数据通常可以由线图和点图进行可视化。第 8 章将详细探讨金融时间序列数据。但是，本节中我们依旧使用二维数据集，并介绍一些对金融应用程序有用的替代可视化的方法。

我们要介绍的第一种图表是散点图，它将其中一个数据集的值作为其他数据集的 x 值。图 7-13 展示了散点图。例如，散点图类型可用于绘制一个金融时间序列的收益和另一个时间序列收益的对比。在下面的例子中，我们将对二维数据集和其他一些数据应用散点图：

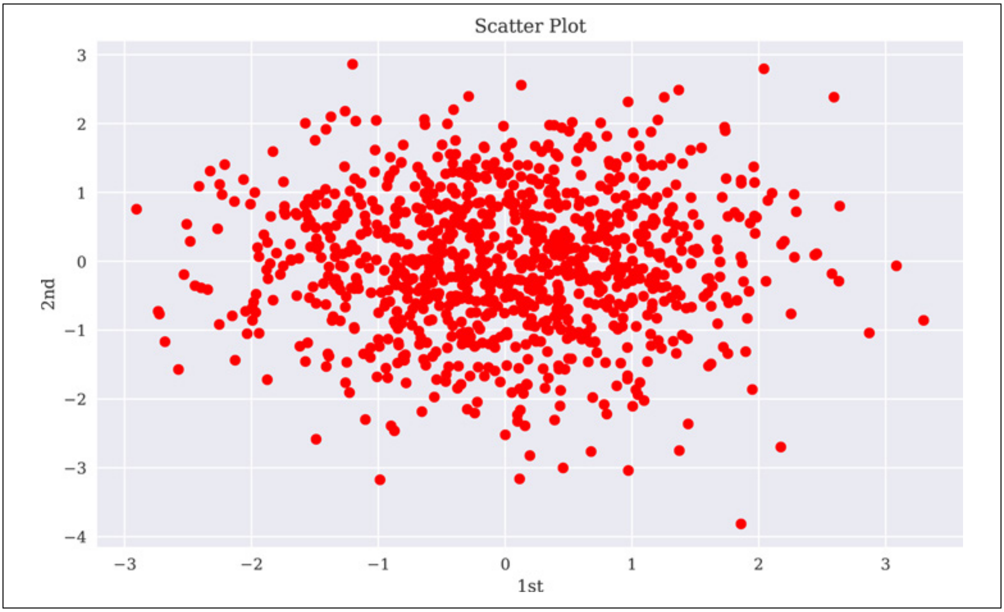


图 7-13　用 `plot()` 函数绘制散点图

```
In [24]: y = np.random.standard_normal((1000, 2)) ❶

In [25]: plt.figure(figsize=(10, 6))
         plt.plot(y[:, 0], y[:, 1], 'ro') ❷
         plt.xlabel('1st')
         plt.ylabel('2nd')
         plt.title('Scatter Plot');
```

❶ 创建一个包含随机数的较大数据集。

❷ 通过 `plt.plot()` 函数制作散点图。

matplotlib 还提供了生成散点图的一个特殊函数。该函数的工作方式与刚才介绍的本质上相同，但是提供了一些额外的功能。图 7-14 展示了与图 7-13 对应的散点图，这次使用的是 `plt.scatter()`函数：

```
In [26]: plt.figure(figsize=(10, 6))
         plt.scatter(y[:, 0], y[:, 1], marker='o') ❶
         plt.xlabel('1st')
         plt.ylabel('2nd')
         plt.title('Scatter Plot');
```

❶ 通过 `plt.scatter()`函数制作散点图。

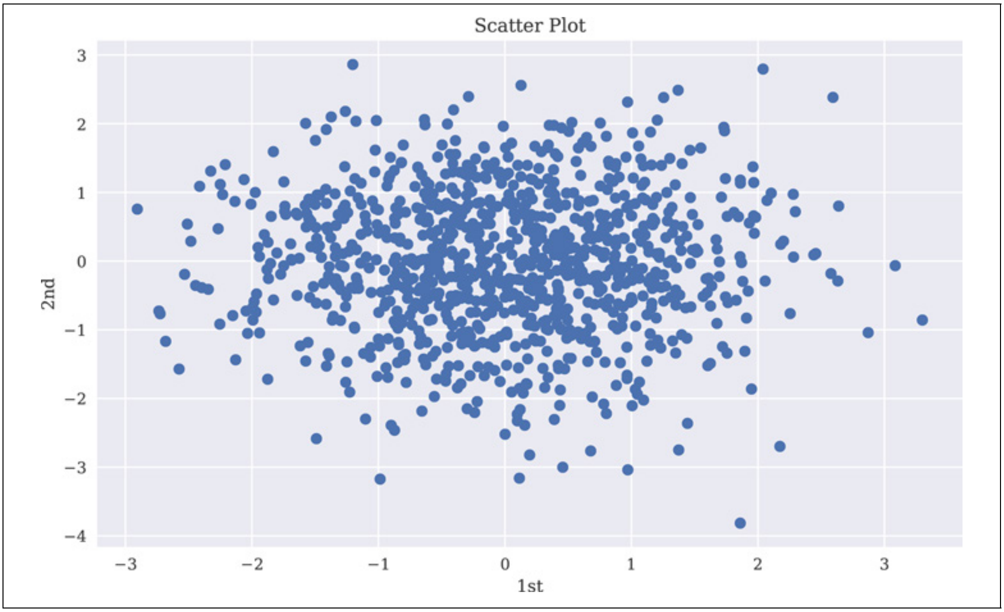


图 7-14　通过 scatter()函数绘制散点图

`plt.scatter()`绘图函数允许加入第三个维度，通过不同的颜色来进行可视化，并使用色卡条加以描述。图 7-15 展示了一个第三维由不同颜色的点表示、图例用色卡条表示的散点图。为此，我们用随机数据生成第三个数据集，这次使用的是 0～10 的整数：

```
In [27]: c = np.random.randint(0, 10, len(y))

In [28]: plt.figure(figsize=(10, 6))
         plt.scatter(y[:, 0], y[:, 1],
                     c=c, ❶
                     cmap='coolwarm', ❷
                     marker='o') ❸
```

```
plt.colorbar()
plt.xlabel('1st')
plt.ylabel('2nd')
plt.title('Scatter Plot');
```

❶ 包含第 3 个数据集。

❷ 选择颜色映射。

❸ 定义散点形状为粗点。

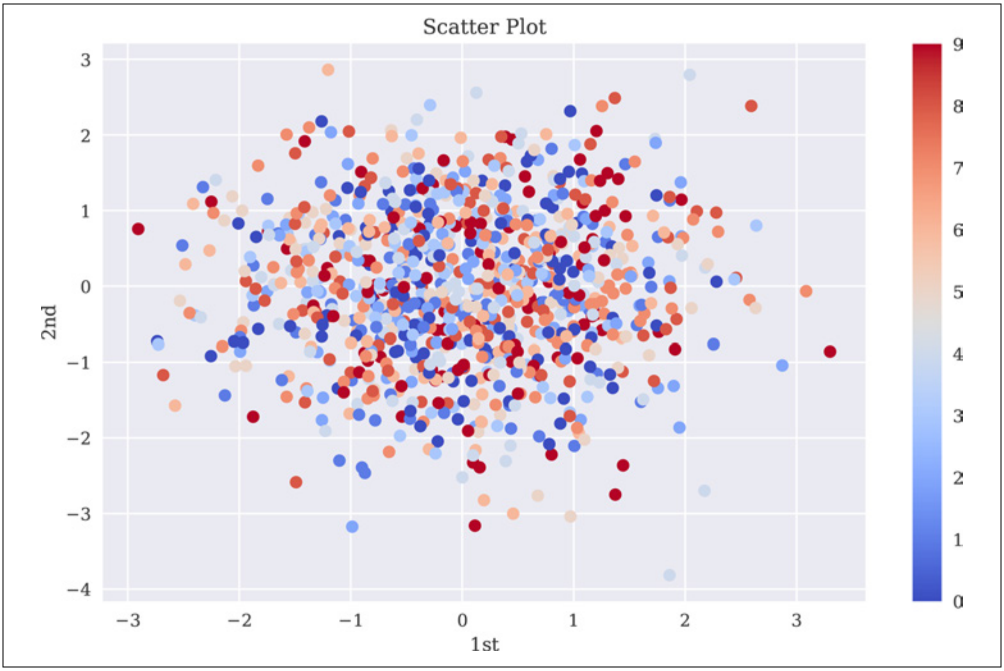


图 7-15　具备第三维的散点图

另一种图表类型——直方图也常常用于金融收益中。图 7-16 在同一个图表中放置了两个数据集的频率值：

```
In [29]: plt.figure(figsize=(10, 6))
         plt.hist(y, label=['1st', '2nd'], bins=25) ❶
         plt.legend(loc=0)
         plt.xlabel('value')
         plt.ylabel('frequency')
         plt.title('Histogram');
```

❶ 通过 `plt.hist()` 函数制作直方图。

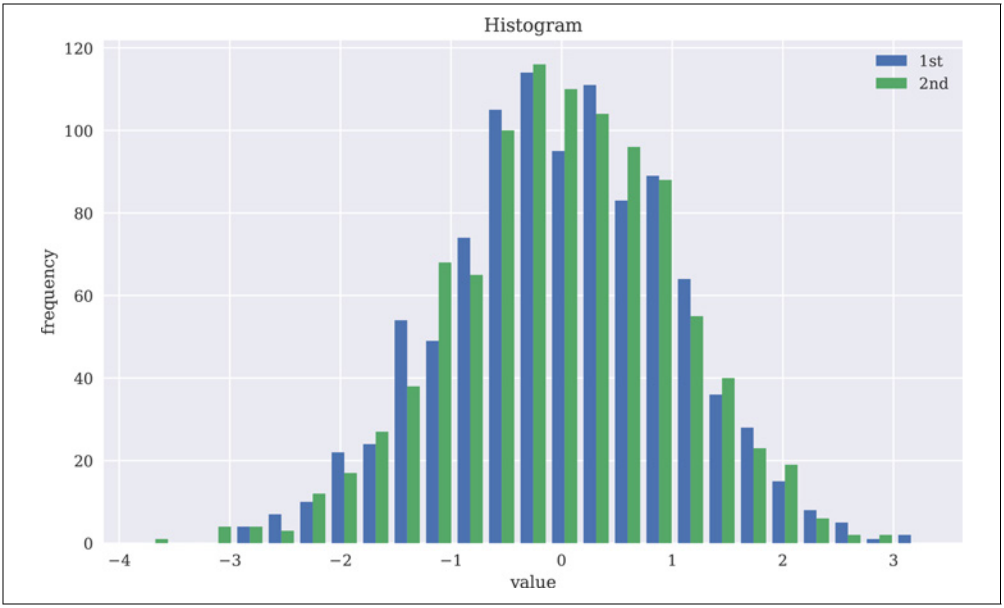


图 7-16　两个数据集的直方图

直方图是金融应用中的重要图表类型，我们要更认真地观察 `plt.hist()`的使用方法。下面的例子说明了 `plt.hist()`所支持的参数：

```
plt.hist(x, bins=10, range=None, normed=False, weights=None, cumulative=False,
bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None,
log=False, color=None, label=None, stacked=False, hold=None, **kwargs)
```

表 7-5 提供了 `plt.hist()`函数主要参数的简单介绍。

表 7-5　**plt.hist()参数**

参数	描述
`x`	列表对象，`ndarray` 对象
`bins`	数据组（bin）数
`range`	数据组的下界和上界
`normed`	规范化为整数 1
`weights`	*x* 轴上每个值的权重
`cumulative`	每个数据组包含的较低组别的数量
`histtype`	选项（字符串）：`bar`、`barstacked`、`step`、`stepfilled`
`align`	选项（字符串）：`left`、`mid`、`right`

续表

参数	描述
orientation	选项（字符串）：horizontal、vertical
rwidth	条的相对宽度
log	对数刻度
color	每个数据集的颜色（类似数组）
label	标签所用的字符串或者字符串序列
stacked	堆叠多个数据集

图 7-17 展示了一个直方图。在图 7-17 中，两个数据集的数据在直方图中堆叠：

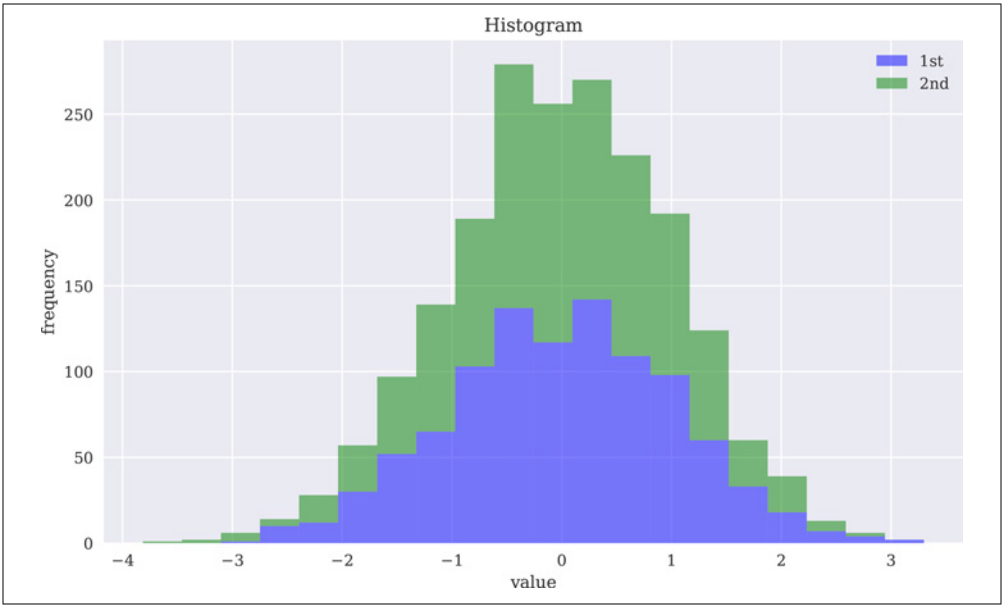


图 7-17　两个数据集堆叠的直方图

```
In [30]: plt.figure(figsize=(10, 6))
         plt.hist(y, label=['1st', '2nd'], color=['b', 'g'],
                     stacked=True, bins=20, alpha=0.5)
         plt.legend(loc=0)
         plt.xlabel('value')
         plt.ylabel('frequency')
         plt.title('Histogram');
```

还有一种实用的图表类型是箱形图。和直方图类似，箱形图可以简洁地展现数据集的

特性，很容易比较多个数据集。图 7-18 根据了根据我们的数据集绘制的箱形图：

```
In [31]: fig, ax = plt.subplots(figsize=(10, 6))
         plt.boxplot(y) ❶
         plt.setp(ax, xticklabels=['1st', '2nd']) ❷
         plt.xlabel('data set')
         plt.ylabel('value')
         plt.title('Boxplot');
```

❶ 通过 `plt.boxplot()` 函数制作箱形图。

❷ 设置单独的 *x* 轴标签。

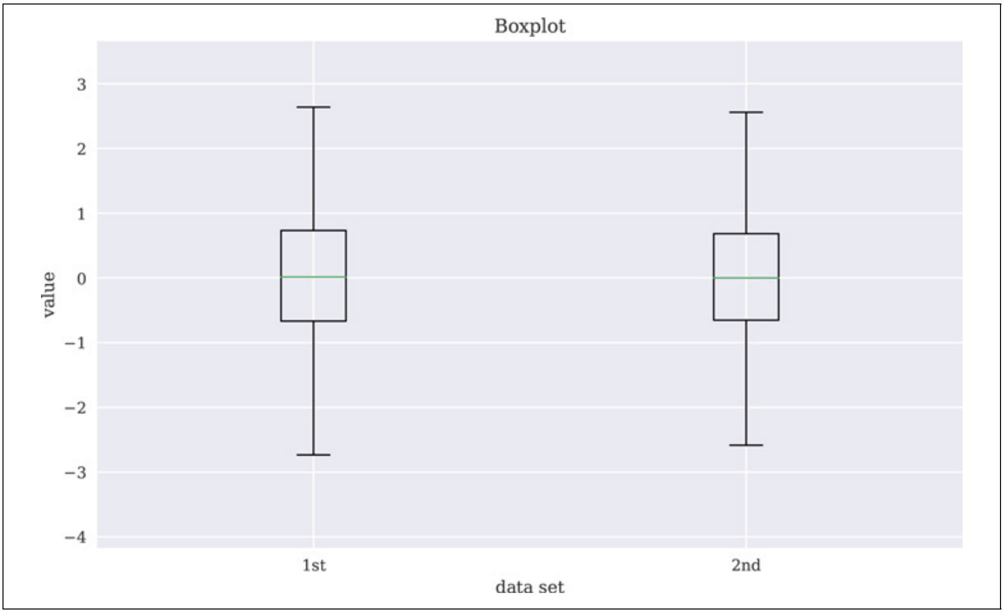


图 7-18　两个数据集的箱形图

最后一个例子将介绍 `plt.setp()` 函数，该函数可以设置（一组）图表实例的属性。例如，考虑下列代码生成的线图：

```
line = plt.plot(data, 'r')
```

使用如下的代码：

```
plt.setp(line, linestyle='--')
```

将线的样式更改为“短画线”。这样，你可以轻松地在图表实例（“艺术家对象”）生成之后修改参数。

在本节的最后一个示例中，我们考虑一个受到数学启迪的图表，这个例子也可以在

matplotlib 的“展厅”中找到。它绘制一个函数的图像，并以图形的方式说明了在某个下限和上限之间函数图像下方区域的面积。换言之，从下限到上限的函数积分值。积分（值）可以表示为 $\int_a^b f(x)\mathrm{d}x$ ，其中 $f(x)=\frac{1}{2}\cdot \mathrm{e}^x+1$，$a=\frac{1}{2}$，$b=\frac{3}{2}$ 。图 7-19 展示了结果图，说明 matplotlib 能够无缝地处理 LaTeX 字体设置，并在图表中加入数学公式。首先定义积分上下限为变量的函数，以及包含 x 和 y 值的数据集：

```
In [32]: def func(x):
             return 0.5 * np.exp(x) + 1 ❶
         a, b = 0.5, 1.5 ❷
         x = np.linspace(0, 2) ❸
         y = func(x) ❹
         Ix = np.linspace(a, b) ❺
         Iy = func(Ix) ❻
         verts = [(a, 0)] + list(zip(Ix, Iy)) + [(b, 0)] ❼
```

❶ 函数定义。

❷ 积分上下限。

❸ 绘制函数图表的 x 值。

❹ 绘制函数图表的 y 值。

❺ 积分上下限内的 x 值。

❻ 积分上下限内的 y 值。

❼ 列表对象，包含表示所绘制多边形坐标的元组对象。

其次是如何绘图。要明确地放置许多单独对象，这有一点难度：

```
In [33]: from matplotlib.patches import Polygon
         fig, ax = plt.subplots(figsize=(10, 6))
         plt.plot(x, y, 'b', linewidth=2) ❶
         plt.ylim(bottom=0) ❷
         poly = Polygon(verts, facecolor='0.7', edgecolor='0.5') ❸
         ax.add_patch(poly) ❸
         plt.text(0.5 * (a + b), 1, r'$\int_a^b f(x)\mathrm{d}x$',
                  horizontalalignment='center', fontsize=20) ❹
         plt.figtext(0.9, 0.075, '$x$') ❺
         plt.figtext(0.075, 0.9, '$f(x)$') ❺
         ax.set_xticks((a, b)) ❻
         ax.set_xticklabels(('$a$', '$b$')) ❻
         ax.set_yticks([func(a), func(b)]) ❼
         ax.set_yticklabels(('$f(a)$', '$f(b)$')); ❼
```

❶ 将函数值绘制为一条蓝色线。

❷ 为纵轴定义最小 *y* 值。

❸ 将多边形（积分区域）绘制为灰色。

❹ 在图中放置积分公式。

❺ 放置轴标签。

❻ 放置 *x* 轴刻度标签。

❼ 放置 *y* 轴刻度标签。

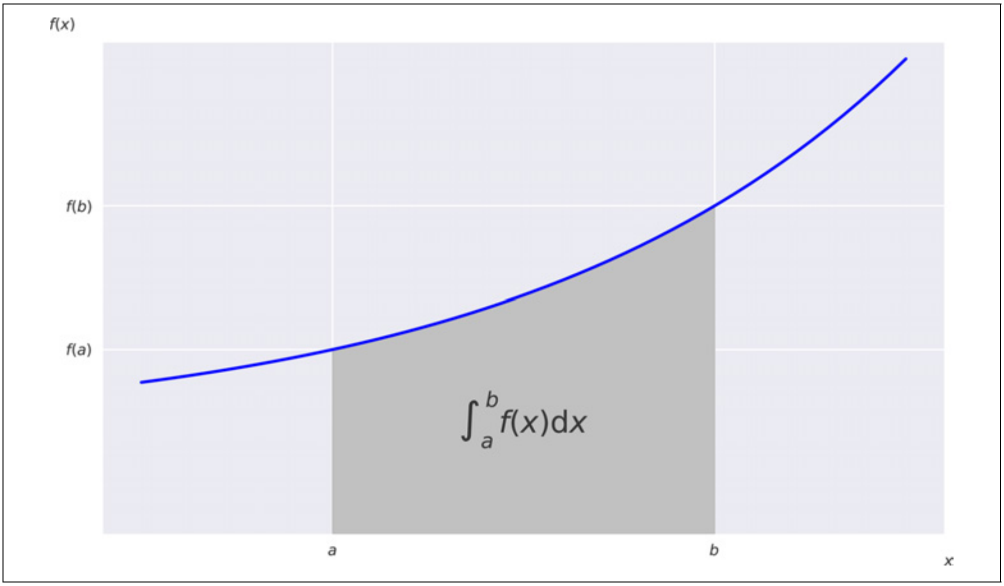


图 7-19　指数函数、积分区域和 LaTeX 标签

7.2　静态 3D 绘图

金融行业中没有很多领域能真正地从三维可视化得益。但是，其中一个得益的应用领域是波动率曲面，它同时显示不同到期日、行权价的交易期权的隐含波动率。附录 B 中有一个欧式看涨期权价值曲面和 vega 值曲面的可视化示例。下文的代码人为地生成了一个类似波动率曲面的图表。考虑如下参数：

- 行权价为 50～150；
- 到期时间为 0.5 年～2.5 年。

上列参数提供了一个二维坐标系。NumPy 的 `np.meshgrid()` 函数可以通过两个一维

ndarray 对象生成此类坐标系：

```
In [34]: strike = np.linspace(50, 150, 24) ❶

In [35]: ttm = np.linspace(0.5, 2.5, 24) ❷

In [36]: strike, ttm = np.meshgrid(strike, ttm) ❸

In [37]: strike[:2].round(1) ❸
Out[37]: array([[ 50. , 54.3, 58.7, 63. , 67.4, 71.7, 76.1, 80.4, 84.8,
                  89.1, 93.5, 97.8, 102.2, 106.5, 110.9, 115.2, 119.6, 123.9,
                 128.3, 132.6, 137. , 141.3, 145.7, 150. ],
                [ 50. , 54.3, 58.7, 63. , 67.4, 71.7, 76.1, 80.4, 84.8,
                  89.1, 93.5, 97.8, 102.2, 106.5, 110.9, 115.2, 119.6, 123.9,
                 128.3, 132.6, 137. , 141.3, 145.7, 150. ]])

In [38]: iv = (strike - 100) ** 2 / (100 * strike) / ttm ❹

In [39]: iv[:5, :3] ❹
Out[39]: array([[1. , 0.76695652, 0.58132045],
               [0.85185185, 0.65333333, 0.4951989 ],
               [0.74193548, 0.56903226, 0.43130227],
               [0.65714286, 0.504 , 0.38201058],
               [0.58974359, 0.45230769, 0.34283001]])
```

❶ 包含行权价的 ndarray 对象。

❷ 包含到期日的 ndarray 对象。

❸ 创建的两个二维 ndarray 对象（网格）。

❹ 虚拟的隐含波动率值。

以下代码绘制的图表如图 7-20 所示：

```
In [40]: from mpl_toolkits.mplot3d import Axes3D ❶
         fig = plt.figure(figsize=(10, 6))
         ax = fig.gca(projection='3d') ❷
         surf = ax.plot_surface(strike, ttm, iv, rstride=2, cstride=2,
                                cmap=plt.cm.coolwarm, linewidth=0.5,
                                antialiased=True) ❸
         ax.set_xlabel('strike') ❹
         ax.set_ylabel('time-to-maturity') ❺
         ax.set_zlabel('implied volatility') ❻
         fig.colorbar(surf, shrink=0.5, aspect=5); ❼
```

❶ 导入对应的 3D 绘图功能，这是必要的，但我们不会直接使用 Axe3D。

❷ 为 3D 绘图创建画布。

❸ 创建 3D 图表。

❹ 设置 x 轴标签。

❺ 设置 y 轴标签。

❻ 设置 z 轴标签。

❼ 创建色卡条。

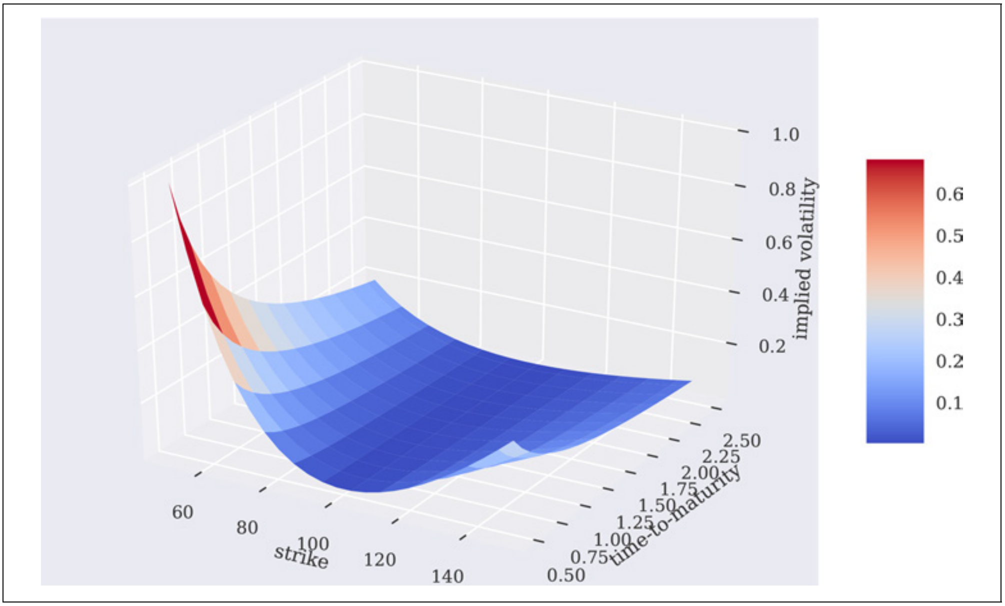


图 7-20 （虚拟）隐含波动率的 3D 曲面图表

表 7-6 提供了 `plt.plot_surface()` 函数的不同参数。

表 7-6 **`plot_surface()`的参数**

参数	描述
`X`、`Y`、`Z`	表示数据值的二维数组
`rstride`	数组行距（步长）
`cstride`	数组列距（步长）
`color`	曲面片颜色
`cmap`	曲面片颜色映射
`facecolors`	单独曲面片的表面颜色

参数	描述
norm	将数值映射到颜色的 Normalize 实例
vmin	映射的最小值
vmax	映射的最大值
shade	表面颜色是否渐变

和二维图表一样，线样式可以由单个点或者下例中的单个三角形表示。图 7-21 用相同的数据绘制 3D 散点图，现在用 view_init()函数来从不同的视角进行设置：

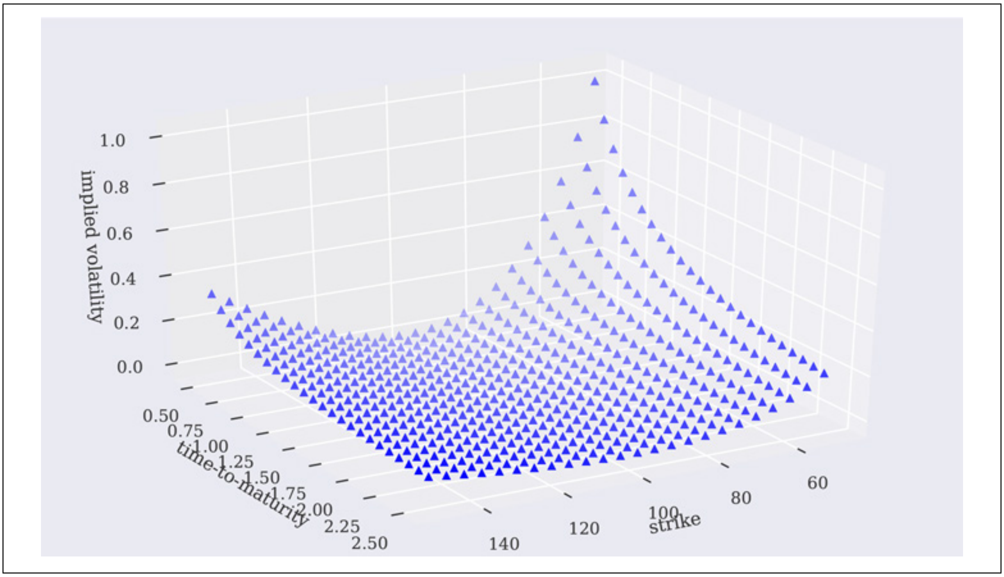


图 7-21　（虚拟）隐含波动率的 3D 散点图

```
In [41]: fig = plt.figure(figsize=(10, 6))
         ax = fig.add_subplot(111, projection='3d')
         ax.view_init(30, 60) ❶
         ax.scatter(strike, ttm, iv, zdir='z', s=25,
                    c='b', marker='^') ❷
         ax.set_xlabel('strike')
         ax.set_ylabel('time-to-maturity')
         ax.set_zlabel('implied volatility');
```

❶ 设置视角。

❷ 创建 3D 散点图。

7.3 交互式 2D 绘图

matplotlib 可以创建静态位图对象或者 PDF 格式的图表。近年来，我们可以用程序库来创建基于 D3.js 标准的交互式图表。这种图表可以缩放、通过鼠标悬停检查数据，还有其他一些功能。一般来说，这种图表很容易嵌入到网页中。

plotly 是一个流行的平台和绘图程序库，专门用于数据科学中的可视化，在数据科学社区中被广泛使用。plotly 的主要优点是与 Python 生态系统紧密集成，易于使用——特别是与 pandas 的 DataFrame 对象和封装器软件包 Cufflinks 相结合时。

为了获取某些功能，需要注册一个 plotly 的免费账户。一旦得到登录凭据，应该将其保存在本地以永久使用。具体细节请参见 *Getting Started with Plotly for Python* 指南。

本节仅专注于一些有选择的方面，其中 Cufflinks 专门用来将保存在 DataFrame 对象中的数据创建为交互式图表。

7.3.1 基本图表

要想从 Jupyter Notebook 环境开始，必须先导入一些程序库，并开启笔记本绘图模式：

```
In [42]: import pandas as pd

In [43]: import cufflinks as cf ❶
In [44]: import plotly.offline as plyo❷

In [45]: plyo.init_notebook_mode(connected=True)❸
```

❶ 导入 `Cufflinks`。

❷ 导入 `plotly` 的离线绘图功能。

❸ 开启笔记本绘图模式。

> **远程或本地渲染**
> plotly 中也有在 plotly 服务器上渲染图表的选项。不过，笔记本模式通常快得多，特别是在处理较大数据集时。话虽如此，某些功能（如 plotly 的流绘图服务）只有通过与服务器的通信才能使用。

下面的例子仍然使用伪随机数，这一次这些伪随机数保存在包含 DatetimeIndex 的 DataFrame 对象中（也就是时间序列数据）：

```
In [46]: a = np.random.standard_normal((250, 5)).cumsum(axis=0) ❶

In [47]: index = pd.date_range('2019-1-1', ❷
```

```
                               freq='B', ❸
                               periods=len(a)) ❹

In [48]: df = pd.DataFrame(100 + 5 * a, ❺
                           columns=list('abcde'), ❻
                           index=index) ❼
In [49]: df.head() ❽
Out[49]:                     a           b          c          d          e
         2019-01-01 109.037535  98.693865 104.474094 96.878857 100.621936
         2019-01-02 107.598242  97.005738 106.789189 97.966552 100.175313
         2019-01-03  101.639668  100.332253 103.183500 99.747869 107.902901
         2019-01-04  98.500363  101.208283 100.966242 94.023898 104.387256
         2019-01-07  93.941632 103.319168 105.674012 95.891062 86.547934
```

❶ 标准正态分布伪随机数。

❷ `DatetimeIndex` 对象的起始日期。

❸ 频率（“交易日”）。

❹ 所需期数。

❺ 原始数据的线性变换。

❻ 单字符的列标题。

❼ `DatetimeIndex` 对象。

❽ 前 5 行数据。

Cufflinks 为 `DataFrame` 添加了一个新方法：`df.iplot()`。这个方法在后端使用 plotly 创建交互式图表。本节的代码示例全部使用该方法，从而以静态位图的形式下载交互式图表，然后将其嵌入到文本中。在 Jupyter Notebook 环境中，创建的图表都是交互式的。以下代码的执行结果如图 7-22 所示：

```
In [50]: plyo.iplot( ❶
             df.iplot(asFigure=True), ❷
             # image='png', ❸
             filename='ply_01' ❹
         )
```

❶ 这利用了 plotly 的离线（笔记本模式）功能。

❷ 以 `asFigure=True` 为参数调用 `df.iplot()`方法，可以进行本地绘图和嵌入。

❸ `image` 选项还提供了图表的静态位图版本。

❹ 指定保存的位图文件名（文件扩展名自动添加）。

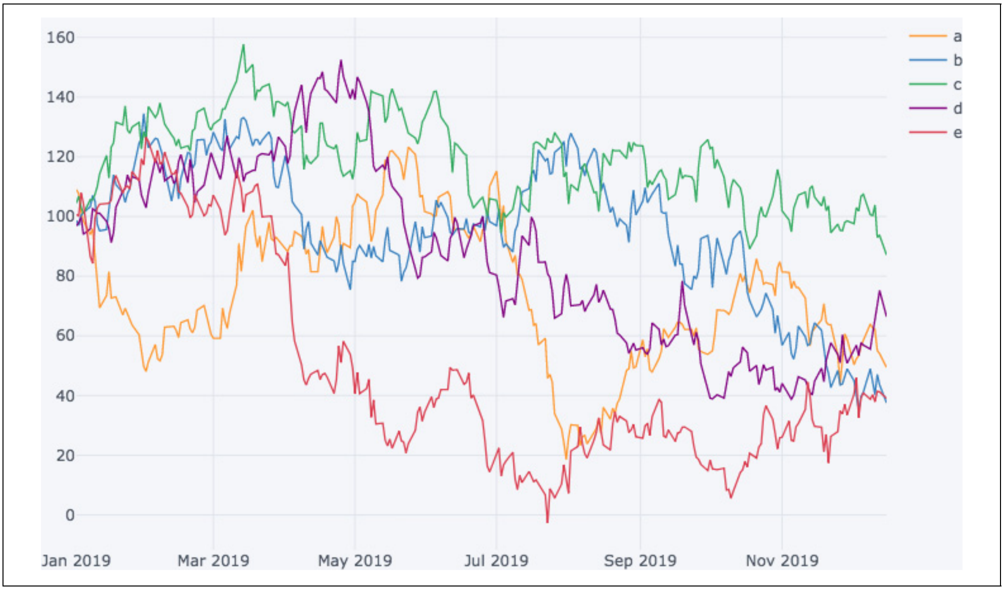


图 7-22　用 plotly、pandas 和 Cufflinks 绘制时间序列数据的线图

和 matplotlib 及 pandas 的绘图功能一样，`df.iplot()`可以用多种参数定制图表（结果见图 7-23）：

```
In [51]: plyo.iplot(
             df[['a', 'b']].iplot(asFigure=True,
                      theme='polar', ❶
                      title='A Time Series Plot', ❷
                      xTitle='date', ❸
                      yTitle='value', ❹
                      mode={'a': 'markers', 'b': 'lines+markers'}, ❺
                      symbol={'a': 'circle', 'b': 'diamond'}, ❻
                      size=3.5, ❼
                      colors={'a': 'blue', 'b': 'magenta'}, ❽
                                 ),
             # image='png',
             filename='ply_02'
         )
```

❶ 为图表选择主题（绘图样式）。

❷ 添加标题。

❸ 添加 x 轴标签。

❹ 添加 y 轴标签。

❺ 按列定义绘图模式（线图、散点图等）。

❻ 按列定义作为散点的符号。

❼ 设定所有散点的大小。

❽ 按列指定绘图颜色。

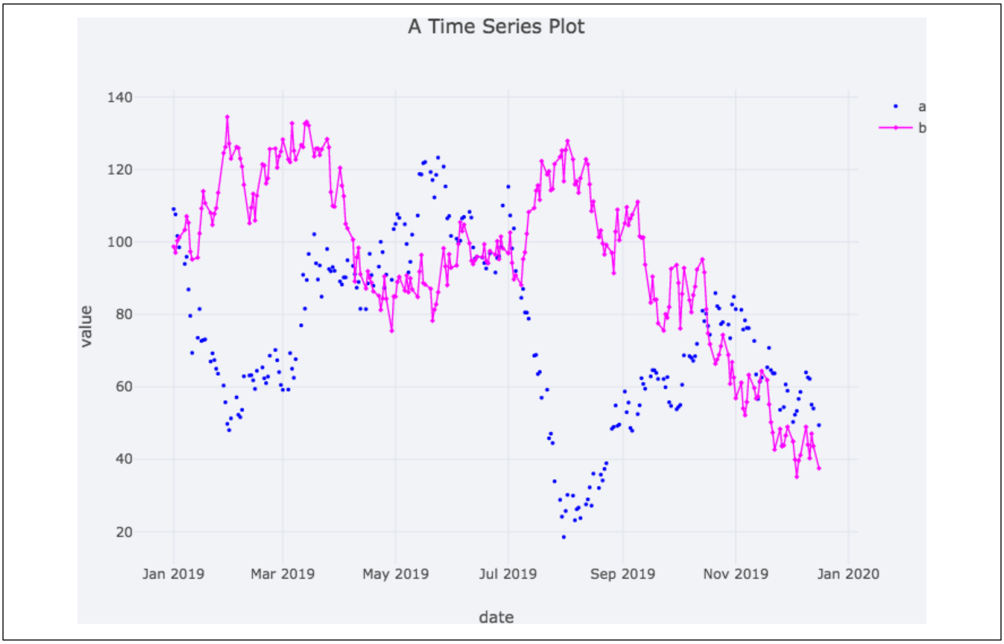


图 7-23　DataFrame 对象两列的线图（定制）

与 matplotlib 类似，plotly 允许使用不同的绘图类型。通过 Cufflinks 使用的绘图类型包括 chart、scatter、bar、box、spread、ratio、heat map、surface、histogram、bubble、bubble3d、scatter3d、scattergeo、ohlc、candle、pie 和 choropleth。接下来介绍一种不同于线图的绘图类型——直方图（见图 7-24）：

```
In [52]: plyo.iplot(
             df.iplot(kind='hist', ❶
                      subplots=True, ❷
                      bins=15, ❸
                      asFigure=True),
             # image='png',
             filename='ply_03'
         )
```

❶ 指定绘图类型。

❷ 要求每列有单独的子图。

❸ 设置 bins 参数（使用的数据组等于绘制的柱形数量）。

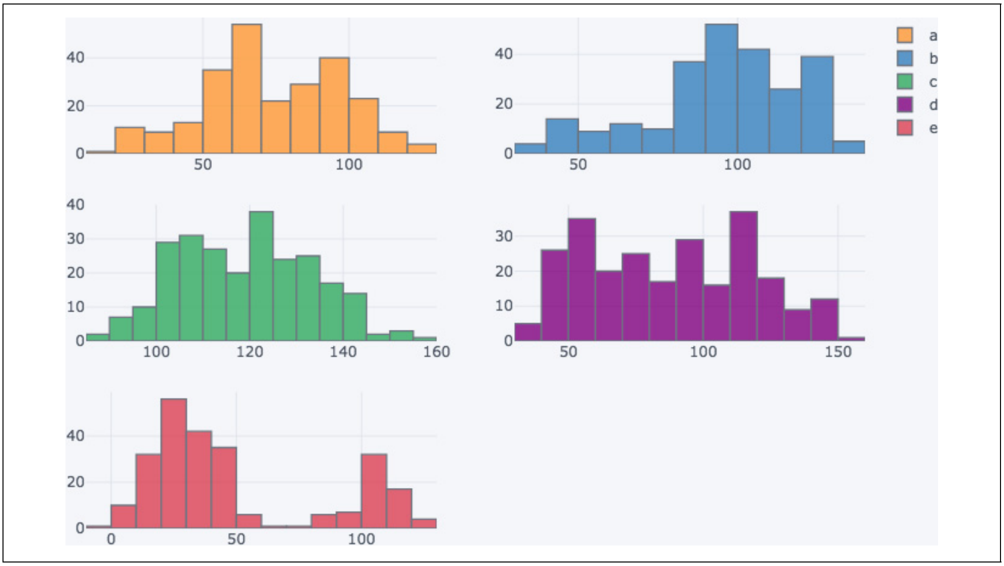


图 7-24　**DataFrame** 对象各列的直方图

7.3.2　金融图表

事实证明，plotly、Cufflinks 和 pandas 的组合在处理金融时间序列数据时特别强大。Cufflinks 提供创建典型金融图表的功能，并增加了典型的金融图表元素，例如相对强弱指数（RSI）等。为此，Cufflinks 创建了一个 `QuantFig` 对象，Cufflinks 可以像 `DataFrame` 对象那样根据 `QuantFig` 绘制图表。

本节使用真实的金融数据集、欧元/美元汇率的时间序列数据（来源：福汇集团）来进行讲解：

```
In [54]: raw = pd.read_csv('../../source/fxcm_eur_usd_eod_data.csv',
                           index_col=0, parse_dates=True) ❶

In [55]: raw.info() ❷
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 1547 entries, 2013-01-01 22:00:00 to
         2017-12-31 22:00:00
         Data columns (total 8 columns):
         BidOpen      1547 non-null float64
         BidHigh      1547 non-null float64
         BidLow       1547 non-null float64
         BidClose     1547 non-null float64
         AskOpen      1547 non-null float64
         AskHigh      1547 non-null float64
         AskLow       1547 non-null float64
         AskClose     1547 non-null float64
         dtypes: float64(8)
```

```
memory usage: 108.8 KB

In [56]: quotes = raw[['AskOpen', 'AskHigh', 'AskLow', 'AskClose']] ❸
         quotes = quotes.iloc[-60:] ❹
         quotes.tail() ❺
Out[56]:                      AskOpen  AskHigh   AskLow  AskClose
         2017-12-25 22:00:00  1.18667  1.18791  1.18467   1.18587
         2017-12-26 22:00:00  1.18587  1.19104  1.18552   1.18885
         2017-12-27 22:00:00  1.18885  1.19592  1.18885   1.19426
         2017-12-28 22:00:00  1.19426  1.20256  1.19369   1.20092
         2017-12-31 22:00:00  1.20092  1.20144  1.19994   1.20144
```

❶ 从 CSV 文件中读取金融数据。

❷ 得到的 `DataFrame` 对象包含多列和超过 1500 行的数据。

❸ 从 `DataFrame` 对象中选择 4 列（开盘价、最高价、最低价、收盘价，或者 OHLC）。

❹ 只有几行数据用于可视化。

❺ 返回结果数据集 `quotes` 的最后 5 行。

在实例化期间，`QuantFig` 对象将 `DataFrame` 对象作为输入，可以进行一些基本的定制。然后，使用 `qf.iplot()`方法，按照保存在 `QuantFig` 对象中的数据来绘制图表（见图 7-25）：

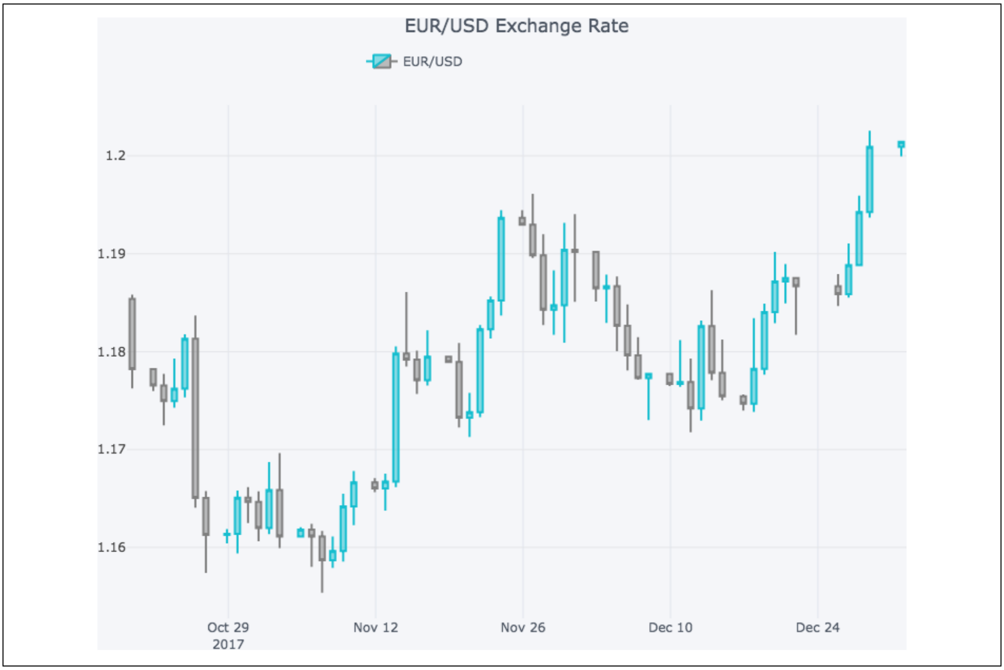


图 7-25　美元/欧元数据的 OHLC 图表

```
In [57]: qf = cf.QuantFig(
                      quotes, ❶
                      title='EUR/USD Exchange Rate', ❷
                      legend='top', ❸
                      name='EUR/USD' ❹
         )
In [58]: plyo.iplot(
                  qf.iplot(asFigure=True),
                  # image='png',
                  filename='qf_01'
         )
```

❶ 传递给 `QuantFig` 对象的用于构造程序的 `DataFrame` 对象。

❷ 添加一个图形标题。

❸ 图例放在图表顶部。

❹ 为数据集指定一个名称。

通过 `QuantFig` 对象的不同方法，我们可以添加典型的金融图表元素，如布林线（见图 7-26）：

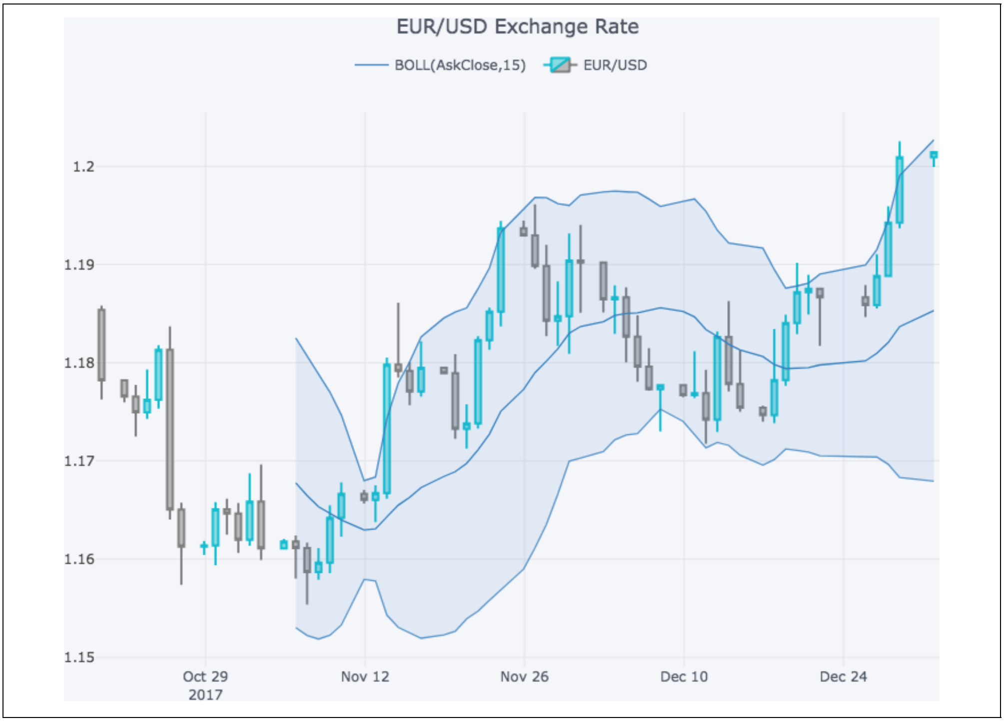


图 7-26　欧元/美元数据的 OHLC 图表（带布林线）

```
In [59]: qf.add_bollinger_bands(periods=15, ❶
                                boll_std=2) ❷

In [60]: plyo.iplot(qf.iplot(asFigure=True),
                  # image='png',
                  filename='qf_02'

         )
```

❶ 布林线周期。

❷ 用于布林线宽度的标准差。

某些金融指标（如 RSI）可以作为子图添加（见图 7-27）：

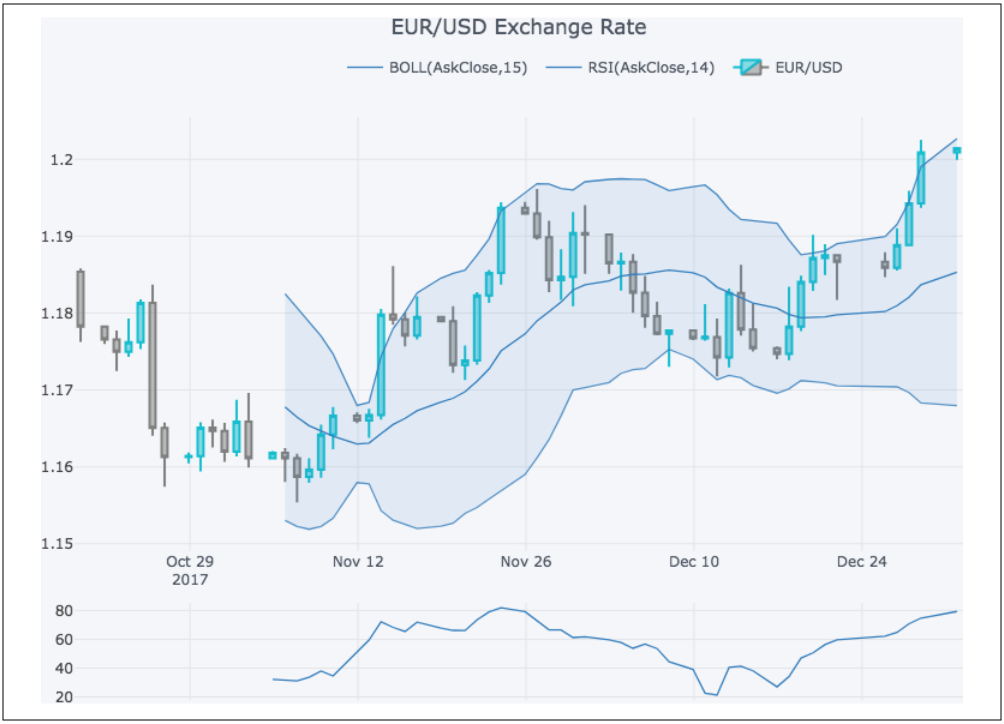


图 7-27　欧元/美元数据的 OHLC 图表（带布林线和 RSI 指标）

```
In [61]: qf.add_rsi(periods=14, ❶
                    showbands=False) ❷

In [62]: plyo.iplot(
                  qf.iplot(asFigure=True),
                  # image='png',
```

```
                    filename='qf_03'
          )
```

❶ 设置 RSI 周期。

❷ 不显示高线或者低线。

7.4 结语

我们可以将 matplotlib 视为 Python 数据可视化的基准和主力，它与 NumPy 和 pandas 紧密集成，其基本功能可以轻松且方便地访问。但是，另一方面，matplotlib 是相当强大的库，具有复杂的 API，本章中无法概述 matplotlib 的所有功能。

本章介绍了 matplotlib 可用于许多金融环境下的基本 2D 和 3D 绘图函数。其他章节提供使用这个可视化基本库的进一步示例。

此外，本章还介绍了 plotly 和 Cufflinks 的组合。这一组合可以方便地创建 D3.js 交互图表，一般来说，只需在 DataFrame 对象上调用一个方法就能实现。所有技术细节都由后端完成。而且，Cufflinks 以 QuantFig 对象的形式，提供了轻松创建包含流行金融指标的典型金融图表的方式。

7.5 延伸阅读

网上可以找到各种各样的 matplotlib 资源：

- matplotlib 主页也许是最好的起点；
- matplotlib 网站上有一个展厅，其中有许多实用示例；
- matplotlib 官网中的 2D 绘图教程；
- matplotlib 官网中的 3D 绘图教程。

访问展厅已经成为学习 matplotlib 的必备过程，在那里可以看到相应的可视化示例，并从对应的示例代码开始工作。

plotly 和 Cufflinks 软件包的主要资源也在网上：

- plotly 主页；
- 用于 Python 的 plotly 教程；
- Cufflinks GitHub 网页。

第 8 章

金融时间序列

时间是阻止所有事情同时发生的力量。

——雷·卡明斯

金融学中最重要的数据类型之一是金融时间序列。它是以日期和/或时间作为索引的数据。例如，股价就是金融时间序列数据。类似地，欧元/美元汇率也是金融时间序列数据；汇率在短暂的时间间隔内报价，一组此类报价就是汇率的时间序列。

没有一种金融学科不将时间作为重要因素考虑，这和物理学及其他科学相同。Python 中处理时间序列的主要工具是 pandas 库。pandas 的主要作者 Wes McKinney 在大型对冲基金 AQR 资本管理公司任分析师时开始开发这个库。可以肯定地说，pandas 从设计的一开始就是为了处理金融时间序列数据。

本章主要基于两组逗号分隔文件（CSV）形式的金融时间序列数据，按以下顺序介绍。

金融数据

本节介绍用 pandas 处理金融时间序列数据的基础知识：数据导入、得出汇总统计量、随时计算变化以及重新采样。

滚动统计

在金融分析中，滚动统计起到了重要的作用，这一般是时间间隔固定、并在整个数据集中向前滚动计算得出的统计量。常见的例子是简单移动平均数。本节介绍 pandas 对此类统计量计算的支持。

相关分析

本节介绍一个基于金融时间序列数据的标普 500 股票指数和 VIX 波动率指数的案例。金融时间序列数据为这两种指数负相关这一典型（经验）事实提供了某种支持。

高频数据

本节使用金融业已经很常见的高频数据（或分笔报价数据）进行讲解。pandas 在对此类数据集的处理中再次证明了自己的强大。

8.1 金融数据

本节使用的是以 CSV 文件形式在本地存储的金融数据集。从技术上讲，CSV 文件是包含数据行结构的文本文件，其特征是以逗号分隔单个值。在导入数据之前，导入一些软件包并进行定制：

```
In [1]: import numpy as np
        import pandas as pd
        from pylab import mpl, plt
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

8.1.1 数据导入

pandas 提供不同的函数和 DataFrame 方法，以导入不同存储格式（CSV、SQL、Excel 等）的数据，并将数据导出为不同格式（详见第 9 章）。下面的代码通过 `pd.read_csv()` 函数导入 CSV[1]文件中的时间序列数据：

```
In [2]: filename = '../../source/tr_eikon_eod_data.csv' ❶

In [3]: f = open(filename, 'r') ❷
        f.readlines()[:5] ❷
Out[3]: ['Date,AAPL.O,MSFT.O,INTC.O,AMZN.O,GS.N,SPY,.SPX,.VIX,EUR=,XAU=,GDX,
        ,GLD\n',
         '2010-01-01,,,,,,,,,1.4323,1096.35,,\n',
         '2010-01-04,30.57282657,30.95,20.88,133.9,173.08,113.33,1132.99,20.04,
        ,1.4411,1120.0,47.71,109.8\n',
         '2010-01-05,30.625683660000004,30.96,20.87,134.69,176.14,113.63,1136.52,
        ,19.35,1.4368,1118.65,48.17,109.7\n',
         '2010-01-06,30.138541290000003,30.77,20.8,132.25,174.26,113.71,1137.14,
        ,19.16,1.4412,1138.5,49.34,111.51\n']

In [4]: data = pd.read_csv(filename, ❸
                           index_col=0, ❹
                           parse_dates=True) ❺

In [5]: data.info() ❻
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 2216 entries, 2010-01-01 to 2018-06-29
```

1 这个文件包含不同金融工具的日终（EOD）数据，通过 Thomson Reuters Eikon Data API 读取。——原注

```
Data columns (total 12 columns):
AAPL.O    2138 non-null float64
MSFT.O    2138 non-null float64
INTC.O    2138 non-null float64
AMZN.O    2138 non-null float64
GS.N      2138 non-null float64
SPY       2138 non-null float64
.SPX      2138 non-null float64
.VIX      2138 non-null float64
EUR=      2216 non-null float64
XAU=      2211 non-null float64
GDX       2138 non-null float64
GLD       2138 non-null float64
dtypes: float64(12)
memory usage: 225.1 KB
```

❶ 指定路径和文件名。

❷ 显示原始数据（Linux/Mac）的前 5 行。

❸ 传递给 `pd.read_scv()`函数的文件名。

❹ 指定第一列作为索引处理。

❺ 指定索引值为 `datetime` 类型。

❻ 结果为 `DataFrame` 对象。

在这一阶段，金融分析师可能首先观察数据，对其进行检查或者可视化（见图 8-1）：

```
In [6]: data.head() ❶
Out[6]:
                AAPL.O MSFT.O INTC.O AMZN.O   GS.N    SPY    .SPX   .VIX \
    Date
    2010-01-01        NaN    NaN    NaN    NaN    NaN    NaN     NaN    NaN
    2010-01-04  30.572827 30.950  20.88 133.90 173.08 113.33 1132.99 20.04
    2010-01-05  30.625684 30.960  20.87 134.69 176.14 113.63 1136.52 19.35
    2010-01-06  30.138541 30.770  20.80 132.25 174.26 113.71 1137.14 19.16
    2010-01-07  30.082827 30.452  20.60 130.00 177.67 114.19 1141.69 19.06

                  EUR=    XAU=   GDX    GLD
    Date
    2010-01-01  1.4323 1096.35   NaN    NaN
    2010-01-04  1.4411 1120.00 47.71 109.80
    2010-01-05  1.4368 1118.65 48.17 109.70
    2010-01-06  1.4412 1138.50 49.34 111.51
    2010-01-07  1.4318 1131.90 49.10 110.82

In [7]: data.tail() ❷
Out[7]:
             AAPL.O MSFT.O INTC.O  AMZN.O   GS.N    SPY    .SPX  .VIX \
    Date
    2018-06-25  182.17  98.39  50.71 1663.15 221.54 271.00 2717.07 17.33
```

```
2018-06-26  184.43  99.08  49.67 1691.09 221.58 271.60 2723.06 15.92
2018-06-27  184.16  97.54  48.76 1660.51 220.18 269.35 2699.63 17.91
2018-06-28  185.50  98.63  49.25 1701.45 223.42 270.89 2716.31 16.85
2018-06-29  185.11  98.61  49.71 1699.80 220.57 271.28 2718.37 16.09

               EUR=     XAU=    GDX    GLD
Date
2018-06-25  1.1702 1265.00 22.01 119.89
2018-06-26  1.1645 1258.64 21.95 119.26
2018-06-27  1.1552 1251.62 21.81 118.58
2018-06-28  1.1567 1247.88 21.93 118.22
2018-06-29  1.1683 1252.25 22.31 118.65

In [8]: data.plot(figsize=(10, 12), subplots=True); ❸
```

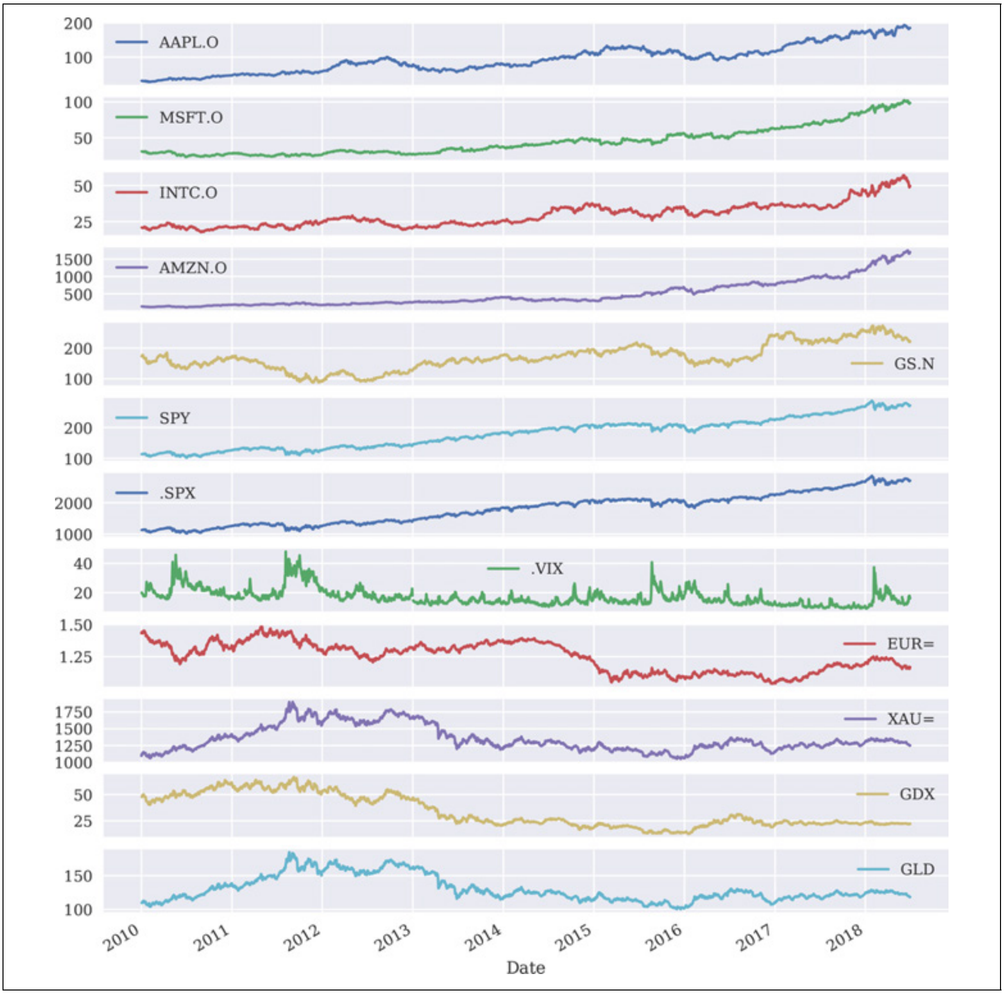


图 8-1　以线图表示的金融时间序列数据

❶ 前 5 行。

❷ 最后 5 行显示。

❸ 这个语句通过多个子图来可视化整个数据集。

这里使用的数据来自 Thomson Reuters (TR) Eikon Data API。TR 金融工具代码称作路透金融工具代码（RIC）。RIC 表示的金融工具为：

```
In [9]: instruments = ['Apple Stock', 'Microsoft Stock',
                       'Intel Stock', 'Amazon Stock', 'Goldman Sachs Stock',
                       'SPDR S&P 500 ETF Trust', 'S&P 500 Index',
                       'VIX Volatility Index', 'EUR/USD Exchange Rate',
                       'Gold Price', 'VanEck Vectors Gold Miners ETF',
                       'SPDR Gold Trust']

In [10]: for ric, name in zip(data.columns, instruments):
             print('{:8s} | {}'.format(ric, name))
         AAPL.O   | Apple Stock
         MSFT.O   | Microsoft Stock
         INTC.O   | Intel Stock
         AMZN.O   | Amazon Stock
         GS.N     | Goldman Sachs Stock
         SPY      | SPDR S&P 500 ETF Trust
         .SPX     | S&P 500 Index
         .VIX     | VIX Volatility Index
         EUR=     | EUR/USD Exchange Rate
         XAU=     | Gold Price
         GDX      | VanEck Vectors Gold Miners ETF
         GLD      | SPDR Gold Trust
```

8.1.2 汇总统计

金融分析师采取的下一个步骤是观察不同的数据集汇总统计，从而对其有个总体的“感觉”：

```
In [11]: data.info() ❶
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2216 entries, 2010-01-01 to 2018-06-29
         Data columns (total 12 columns):
         AAPL.O    2138 non-null float64
         MSFT.O    2138 non-null float64
         INTC.O    2138 non-null float64
         AMZN.O    2138 non-null float64
         GS.N      2138 non-null float64
         SPY       2138 non-null float64
         .SPX      2138 non-null float64
         .VIX      2138 non-null float64
         EUR=      2216 non-null float64
```

```
        XAU=      2211 non-null float64
        GDX       2138 non-null float64
        GLD       2138 non-null float64
        dtypes: float64(12)
        memory usage: 225.1 KB

In [12]: data.describe().round(2) ❷
Out[12]:
            AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N     SPY    .SPX   .VIX \
    Count  2138.00 2138.00 2138.00 2138.00 2138.00 2138.00 2138.00 2138.00
    mean     93.46   44.56   29.36  480.46  170.22  180.32 1802.71   17.03
    std      40.55   19.53    8.17  372.31   42.48   48.19  483.34    5.88
    min      27.44   23.01   17.66  108.61   87.70  102.20 1022.58    9.14
    25%      60.29   28.57   22.51  213.60  146.61  133.99 1338.57   13.07
    50%      90.55   39.66   27.33  322.06  164.43  186.32 1863.08   15.58
    75%     117.24   54.37   34.71  698.85  192.13  210.99 2108.94   19.07
    max     193.98  102.49   57.08 1750.08  273.38  286.58 2872.87   48.00

                EUR=    XAU=     GDX     GLD
    count   2216.00 2211.00 2138.00 2138.00
    mean       1.25 1349.01   33.57  130.09
    std        0.11 188.75    15.17   18.78
    min        1.04 1051.36   12.47  100.50
    25%        1.13 1221.53   22.14  117.40
    50%        1.27 1292.61   25.62  124.00
    75%        1.35 1428.24   48.34  139.00
    max        1.48 1898.99   66.63  184.59
```

❶ `info()`给出 `DataFrame` 对象的相关元信息。

❷ `describe()`提供每列的实用标准统计量。

敏锐的洞察力

pandas 提供了许多方法（如 `info` 和 `describe`），可以获得新导入的金融时间序列数据集的简单概况。它们还能快捷地检查导入程序是否按照要求进行（例如，`DataFrame` 对象是否真正包含 `DatetimeIndex` 类型的索引）。

当然，pandas 也提供自定义统计类型及显示方式的选项：

```
In [13]: data.mean() ❶
Out[13]: AAPL.O     93.455973
         MSFT.O     44.561115
         INTC.O     29.364192
         AMZN.O    480.461251
```

```
GS.N       170.216221
SPY        180.323029
.SPX      1802.713106
.VIX        17.027133
EUR=         1.248587
XAU=      1349.014130
GDX         33.566525
GLD        130.086590
dtype: float64

In [14]: data.aggregate([min, ❷
                         np.mean, ❸
                         np.std, ❹
                         np.median, ❺
                         max] ❻
         ).round(2)
Out[14]:
        AAPL.O MSFT.O INTC.O  AMZN.O   GS.N    SPY    .SPX  .VIX EUR= \
min      27.44  23.01  17.66  108.61  87.70 102.20 1022.58  9.14  1.04
mean     93.46  44.56  29.36  480.46 170.22 180.32 1802.71 17.03  1.25
std      40.55  19.53   8.17  372.31  42.48  48.19  483.34  5.88  0.11
median   90.55  39.66  27.33  322.06 164.43 186.32 1863.08 15.58  1.27
max     193.98 102.49  57.08 1750.08 273.38 286.58 2872.87 48.00  1.48

           XAU=   GDX    GLD
min     1051.36 12.47 100.50
mean    1349.01 33.57 130.09
std      188.75 15.17  18.78
median  1292.61 25.62 124.00
max     1898.99 66.63 184.59
```

❶ 每列均值。

❷ 每列最小值。

❸ 每列均值。

❹ 每列标准差。

❺ 每列中位数。

❻ 每列最大值。

使用`aggregate()`方法还可以传递自定义函数。

8.1.3 随时间推移的变化

统计分析方法往往基于随时间推移的变化，而不是绝对值。计算时间序列中的随时变

化有多种选择，包括绝对偏差、变化率和对数回报率。

首先介绍绝对偏差，pandas 为此提供了一个特殊的方法：

```
In [15]: data.diff().head() ❶
Out[15]:
              AAPL.O MSFT.O INTC.O AMZN.O  GS.N  SPY .SPX  .VIX  EUR= \
    Date
    2010-01-01       NaN    NaN    NaN    NaN   NaN  NaN  NaN   NaN     NaN
    2010-01-04       NaN    NaN    NaN    NaN   NaN  NaN  NaN   NaN  0.0088
    2010-01-05  0.052857  0.010  -0.01   0.79  3.06 0.30 3.53 -0.69 -0.0043
    2010-01-06 -0.487142 -0.190  -0.07  -2.44 -1.88 0.08 0.62 -0.19  0.0044
    2010-01-07 -0.055714 -0.318  -0.20  -2.25  3.41 0.48 4.55 -0.10 -0.0094

              XAU=  GDX   GLD
    Date
    2010-01-01   NaN   NaN   NaN
    2010-01-04 23.65   NaN   NaN
    2010-01-05 -1.35  0.46 -0.10
    2010-01-06 19.85  1.17  1.81
    2010-01-07 -6.60 -0.24 -0.69
In [16]: data.diff().mean() ❷
Out[16]: AAPL.O    0.064737
         MSFT.O    0.031246
         INTC.O    0.013540
         AMZN.O    0.706608
         GS.N      0.028224
         SPY       0.072103
         .SPX      0.732659
         .VIX     -0.019583
         EUR=     -0.000119
         XAU=      0.041887
         GDX      -0.015071
         GLD      -0.003455
         dtype: float64
```

❶ `diff()`提供两个索引值之间的绝对变化。

❷ 当然，还可以应用聚合运算。

从统计学角度讲，绝对变化不是最优的，因为它们与时间序列数据本身的比例相关。因此，我们通常更偏重变化率。下面的代码可以计算金融环境里的变动率或者回报率（也称为简单回报率），并对其每列的平均值进行可视化（参见图 8-2）：

```
In [17]: data.pct_change().round(3).head() ❶
Out[17]:
                 AAPL.O MSFT.O INTC.O AMZN.O   GS.N   SPY  .SPX   .VIX   EUR= \
    Date
    2010-01-01     NaN     NaN     NaN     NaN     NaN   NaN   NaN     NaN     NaN
    2010-01-04     NaN     NaN     NaN     NaN     NaN   NaN   NaN     NaN   0.006
    2010-01-05   0.002   0.000  -0.000   0.006   0.018 0.003 0.003  -0.034  -0.003
    2010-01-06  -0.016  -0.006  -0.003  -0.018  -0.011 0.001 0.001  -0.010   0.003
    2010-01-07  -0.002  -0.010  -0.010  -0.017   0.020 0.004 0.004  -0.005  -0.007

                  XAU=    GDX    GLD
    Date
    2010-01-01     NaN    NaN    NaN
    2010-01-04   0.022    NaN    NaN
    2010-01-05  -0.001  0.010 -0.001
    2010-01-06   0.018  0.024  0.016
    2010-01-07  -0.006 -0.005 -0.006

In [18]: data.pct_change().mean().plot(kind='bar', figsize=(10, 6)); ❷
```

❶ `pct_change()`计算两个索引值之间的变化率。

❷ 将结果的均值可视化为一个柱状图。

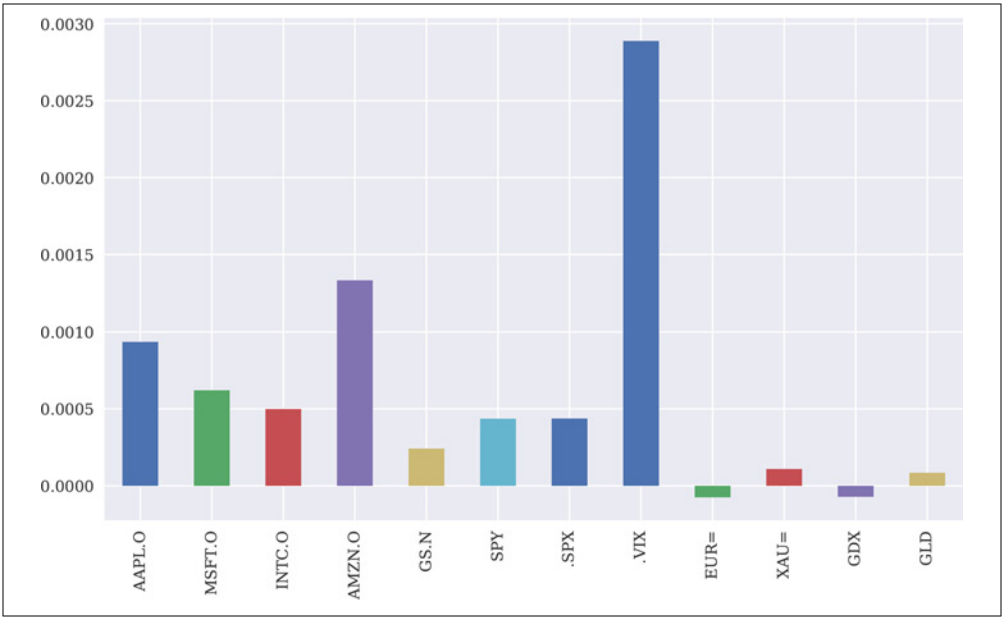


图 8-2　变化率均值柱状图

对数回报率可作为回报率的替代品。在某些情况下，它们更容易处理，因此在金融环

境中往往优先使用对数回报率。[1]

图 8-3 展示了单个金融时间序列的累计对数回报率。这种类型的图表导致了某种形式的规范化：

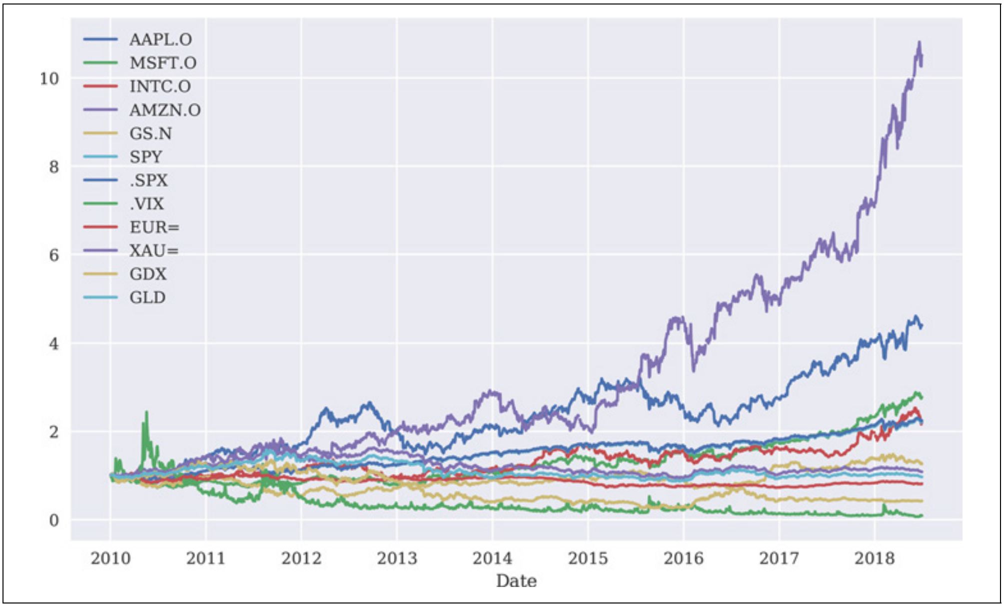


图 8-3　一段时间的累计对数回报率

```
In [19]: rets = np.log(data / data.shift(1))❶

In [20]: rets.head().round(3) ❷
Out[20]:

              AAPL.O MSFT.O INTC.O AMZN.O   GS.N   SPY   .SPX   .VIX EUR= \
  Date
  2010-01-01     NaN    NaN    NaN    NaN    NaN   NaN    NaN    NaN    NaN
  2010-01-04     NaN    NaN    NaN    NaN    NaN   NaN    NaN    NaN  0.006
  2010-01-05   0.002  0.000 -0.000  0.006  0.018 0.003  0.003 -0.035 -0.003
  2010-01-06  -0.016 -0.006 -0.003 -0.018 -0.011 0.001  0.001 -0.010  0.003
  2010-01-07  -0.002 -0.010 -0.010 -0.017  0.019 0.004  0.004 -0.005 -0.007

                XAU=    GDX    GLD
  Date
  2010-01-01     NaN    NaN    NaN
  2010-01-04   0.021    NaN    NaN
```

1　对数回报率的优点之一是可以在一段时间里累加，这对于简单变化率/回报率不适用。——原注

```
2010-01-05 -0.001  0.010 -0.001
2010-01-06  0.018  0.024  0.016
2010-01-07 -0.006 -0.005 -0.006

In [21]: rets.cumsum().apply(np.exp).plot(figsize=(10, 6)); ❸
```

❶ 以向量的方式计算对数回报率。

❷ 结果的一个子集。

❸ 绘制一段时间的累计对数回报率图表；首先调用 `cumsum()`方法，然后对结果应用 `np.exp()`。

8.1.4 重新采样

重新采样是金融时间序列数据的重要操作之一，通常采用向下采样的形式，例如，分笔交易数据序列重新采样的时间间隔为一分钟，也可以将每日观察数据的时间序列重新采样为每周或者每月观察数据（如图 8-4 所示）：

```
In [22]: data.resample('1w', label='right').last().head() ❶
Out[22]:
                AAPL.O MSFT.O INTC.O AMZN.O   GS.N    SPY    .SPX .VIX \
    Date
    2010-01-03       NaN    NaN    NaN    NaN    NaN    NaN     NaN   NaN
    2010-01-10 30.282827  30.66  20.83 133.52 174.31 114.57 1144.98 18.13
    2010-01-17 29.418542  30.86  20.80 127.14 165.21 113.64 1136.03 17.91
    2010-01-24 28.249972  28.96  19.91 121.43 154.12 109.21 1091.76 27.31
    2010-01-31 27.437544  28.18  19.40 125.41 148.72 107.39 1073.87 24.62

                 EUR=    XAU=   GDX    GLD
    Date
    2010-01-03  1.4323 1096.35   NaN    NaN
    2010-01-10  1.4412 1136.10 49.84 111.37
    2010-01-17  1.4382 1129.90 47.42 110.86
    2010-01-24  1.4137 1092.60 43.79 107.17
    2010-01-31  1.3862 1081.05 40.72 105.96

In [23]: data.resample('1m', label='right').last().head() ❷
Out[23]:
                  AAPL.O   MSFT.O INTC.O AMZN.O   GS.N      SPY    .SPX \
    Date
    2010-01-31  27.437544 28.1800 19.40  125.41 148.72 107.3900 1073.87
    2010-02-28  29.231399 28.6700 20.53  118.40 156.35 110.7400 1104.49
    2010-03-31  33.571395 29.2875 22.29  135.77 170.63 117.0000 1169.43
    2010-04-30  37.298534 30.5350 22.84  137.10 145.20 118.8125 1186.69
    2010-05-31  36.697106 25.8000 21.42  125.46 144.26 109.3690 1089.41
```

```
                .VIX    EUR=     XAU=    GDX      GLD
Date
2010-01-31   24.62 1.3862 1081.05 40.72 105.960
2010-02-28   19.50 1.3625 1116.10 43.89 109.430
2010-03-31   17.59 1.3510 1112.80 44.41 108.950
2010-04-30   22.05 1.3295 1178.25 50.51 115.360
2010-05-31   32.07 1.2305 1215.71 49.86 118.881

In [24]: rets.cumsum().apply(np.exp). resample('1m', label='right').last(
                                    ).plot(figsize=(10, 6)); ❸
```

❶ 日终数据以一周为时间间隔重新采样。

❷ 以一月为时间间隔重新采样。

❸ 这就绘制了随时间变化的累计对数回报率图表：首先调用 `cumsum()`，然后对结果应用 `np.exp()`，最后进行重新采样。

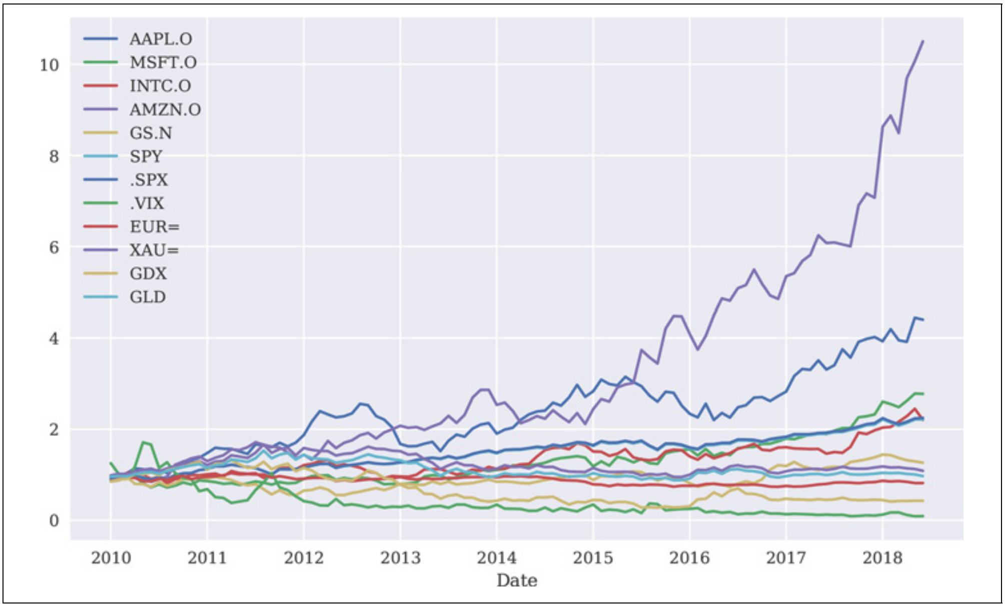


图 8-4　重采样的累计对数回报率图表（每月）



避免预见偏差

在很多情况下，pandas 在重新采样时默认使用区间的左侧标签（或者索引值）。为了在金融业务中保持一致，请确保使用右标签（索引值）——一般是区间内最后一个可用数据点。否则，金融分析中可能潜藏着预见偏差。

8.2　滚动统计

使用滚动统计（也称作金融指标或者金融研究指标）是金融业的传统。例如，滚动统计是金融图表绘制者和技术派交易者的基本工具。本节只使用单个金融时间序列：

```
In [25]: sym = 'AAPL.O'

In [26]: data = pd.DataFrame(data[sym]).dropna()

In [27]: data.tail()
Out[27]:            AAPL.O
         Date
         2018-06-25 182.17
         2018-06-26 184.43
         2018-06-27 184.16
         2018-06-28 185.50
         2018-06-29 185.11
```

8.2.1　概述

用 pandas 计算标准滚动统计量很简单：

```
In [28]: window = 20 ❶

In [29]: data['min'] = data[sym].rolling(window=window).min() ❷

In [30]: data['mean'] = data[sym].rolling(window=window).mean() ❸

In [31]: data['std'] = data[sym].rolling(window=window).std() ❹

In [32]: data['median'] = data[sym].rolling(window=window).median() ❺

In [33]: data['max'] = data[sym].rolling(window=window).max() ❻

In [34]: data['ewma'] = data[sym].ewm(halflife=0.5, min_periods=window).mean() ❼
```

❶ 定义计算窗口，也就是包含的索引值。

❷ 计算滚动最小值。

❸ 计算滚动平均值。

❹ 计算滚动标准差。

❺ 计算滚动中位值。

❻ 计算滚动最大值。

❼ 计算指数加权移动平均值，半衰期为 0.5。

要得到更专业的金融指标，通常需要使用其他软件包（例如，第 7 章中介绍的 Cufflinks 金融图表）。你也可以通过 `apply()` 方法简单地应用定制的方法。

下面的代码显示结果的一个子集，并对一组计算出的滚动统计量进行可视化（见图 8-5）：

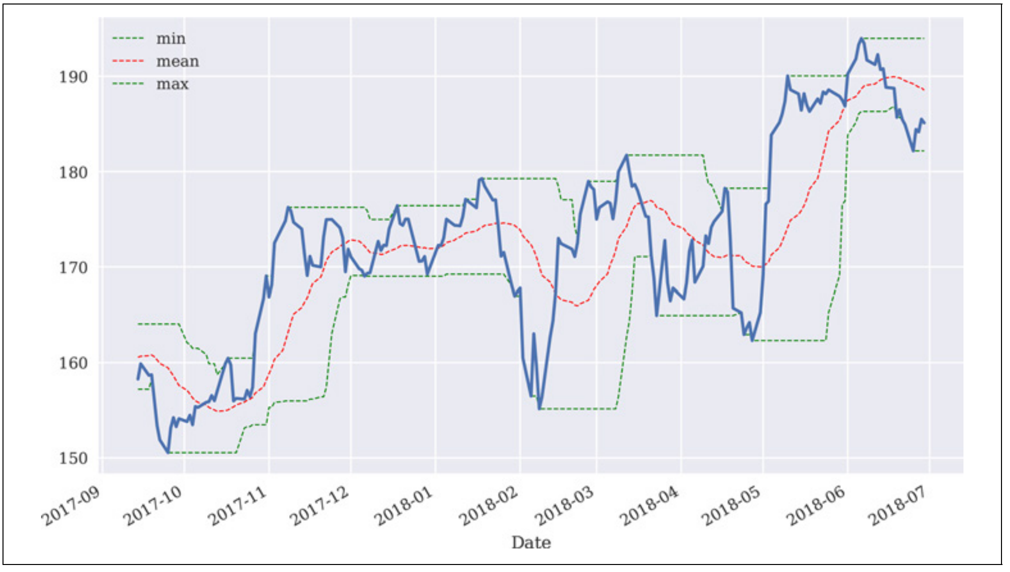


图 8-5　最小值、均值、最大值的滚动统计

```
In [35]: data.dropna().head()
Out[35]:
                AAPL.O        min       mean       std     median       max \
    Date
    2010-02-01 27.818544 27.437544 29.580892 0.933650 29.821542 30.719969
    2010-02-02 27.979972 27.437544 29.451249 0.968048 29.711113 30.719969
    2010-02-03 28.461400 27.437544 29.343035 0.950665 29.685970 30.719969
    2010-02-04 27.435687 27.435687 29.207892 1.021129 29.547113 30.719969
    2010-02-05 27.922829 27.435687 29.099892 1.037811 29.419256 30.719969

                   ewma
    Date
    2010-02-01 27.805432
    2010-02-02 27.936337
    2010-02-03 28.330134
    2010-02-04 27.659299
```

```
    2010-02-05 27.856947

In [36]: ax = data[['min', 'mean', 'max']].iloc[-200:].plot(
             figsize=(10, 6), style=['g--', 'r--', 'g--'], lw=0.8) ❶
         data[sym].iloc[-200:].plot(ax=ax, lw=2.0); ❷
```

❶ 为最后 200 行数据绘制 3 个滚动统计量的图表。

❷ 在图表中添加原始时间序列数据。

8.2.2 技术分析示例

滚动统计是所谓股票“技术分析”中的重要工具之一，技术分析是与“基本面分析”相对的分析方法，后者主要关注所分析股票的财务报表和公司的战略地位。一种基于技术分析的交易策略已经有数十年的历史，它使用两种简单移动平均数（SMA）来实现。具体思路是，交易者应该在短期 SMA 高于长期 SMA 时买进股票（或者金融工具），而在相反的情况下卖出。这些概念可以利用 pandas 和 DataFrame 对象的能力精确地实现。

滚动统计量通常只在窗口（window）参数指定的时期内有足够数据的时候计算。如图 8-6 所示，SMA 时间序列在指定参数有足够数据的那一天才开始：



图 8-6 苹果公司股价和两条简单平均线

```
In [37]: data['SMA1'] = data[sym].rolling(window=42).mean() ❶

In [38]: data['SMA2'] = data[sym].rolling(window=252).mean() ❷

In [39]: data[[sym, 'SMA1', 'SMA2']].tail()
```

```
Out[39]:              AAPL.O        SMA1        SMA2
         Date
         2018-06-25  182.17  185.606190 168.265556
         2018-06-26  184.43  186.087381 168.418770
         2018-06-27  184.16  186.607381 168.579206
         2018-06-28  185.50  187.089286 168.736627
         2018-06-29  185.11  187.470476 168.901032

In [40]: data[[sym, 'SMA1', 'SMA2']].plot(figsize=(10, 6)); ❸
```

❶ 计算短期 SMA 值。

❷ 计算长期 SMA 值。

❸ 可视化股价数据和两个 SMA 时间序列。

在这种背景下，SMA 只是一种手段。它用来算出实施某种交易策略的头寸。图 8-7 中，值 1 表示多头头寸，值-1 表示空头头寸。如果表示 SMA 时间序列的两条线出现交叉，则触发头寸的变化（从视觉上）：



图 8-7　苹果公司股价、两条简单移动平均线及头寸

```
In [41]: data.dropna(inplace=True) ❶

In [42]: data['positions'] = np.where(data['SMA1'] > data['SMA2'], ❷
                                      1, ❸
                                      -1) ❹
In [43]: ax = data[[sym, 'SMA1', 'SMA2', 'positions']].plot(figsize=(10, 6),
                                                    secondary_y='positions')
         ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```

❶ 只保留完整的数据行。

❷ 如果短期 SMA 值大于长期 SMA 值。

❸ 买进股票（加入值 1）。

❹ 否则，卖出股票（加入值–1）。

本例得出的交易策略只能导致少数交易：只有在头寸值变化（两条线发生交叉）时才进行交易。将开始与结束的交易包含在内，总共只进行了 6 次交易。

8.3　相关分析

为了进一步说明 pandas 和金融时间序列数据的使用，考虑标普 500 股票指数和 VIX 波动率指数的情况。典型事实是，当标普 500 指数上升时，VIX 通常下降，反之亦然。这是一种“相关”，而不是因果关系。本节说明如何为“标普 500 与 VIX（高度）负相关”这一典型事实提供某些支持性的统计学证据。[1]

8.3.1　数据

数据集现在包括两个金融时间序列，在图 8-8 中它们都已可视化：

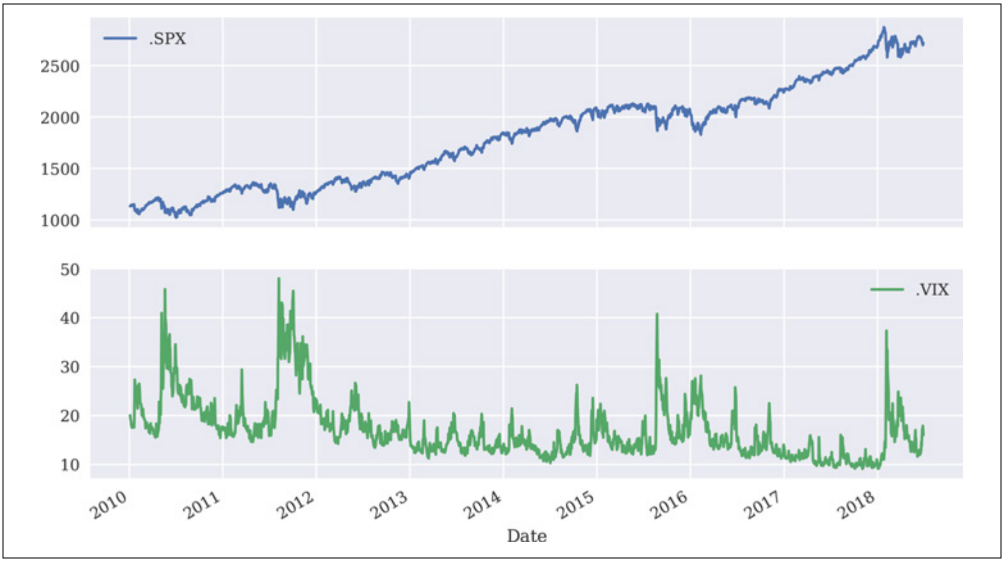


图 8-8　标普 500 和 VIX 时间序列数据（不同子图）

1　造成这一事实的原因之一是，股票指数下跌（例如在一场危机期间），交易量上升，波动率也随之上升。股票指数上涨，投资者通常保持镇静，没有看到太多进行频繁交易的刺激因素。特别是，只做多的投资者此时会努力地进一步利用这个趋势。——原注

```
In [44]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                           index_col=0, parse_dates=True) ❶

In [45]: data = raw[['.SPX', '.VIX']].dropna()

In [46]: data.tail()
Out[46]:                .SPX   .VIX
         Date
         2018-06-25 2717.07 17.33
         2018-06-26 2723.06 15.92
         2018-06-27 2699.63 17.91
         2018-06-28 2716.31 16.85
         2018-06-29 2718.37 16.09

In [47]: data.plot(subplots=True, figsize=(10, 6));
```

❶ 从 CSV 文件读取日终数据（来源于 Thomson Reuters Eikon Data API）。

在单一图表上绘制两个时间序列（的部分），并使用调整的刻度后，通过图 8-9 可明显地看出两种指数之间负相关的典型事实。

```
In [48]: data.loc[:'2012-12-31'].plot(secondary_y='.VIX', figsize=(10, 6));❶
```

❶ `.loc[:DATE]`选择 DATE 值指定日期之前的数据。

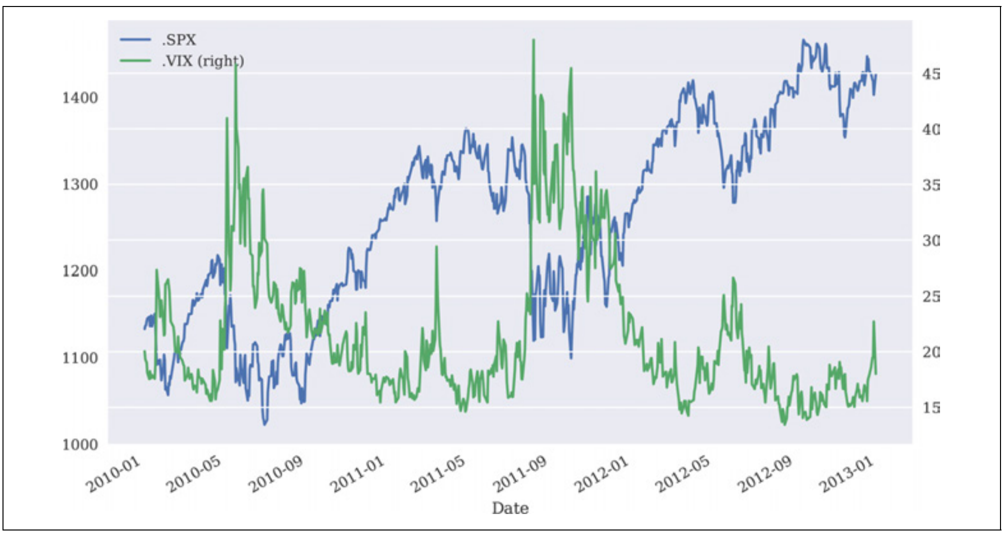


图 8-9　标普 500 和 VIX 时间序列数据（同一图表）

8.3.2　对数回报率

前面已经指出，统计分析通常依靠回报率而不是绝对变化或者绝对价值。因此，我们

将在进行进一步分析之前先计算对数回报率。图 8-10 展示了一段时间里对数回报率的大幅波动。对于这两种指数，都可以发现所谓的“波动聚集性”。一般来说，股票指数的高变动周期与波动率指数中的变动较为同步：

```
In [49]: rets = np.log(data / data.shift(1))

In [50]: rets.head()
Out[50]:                  .SPX      .VIX
         Date
         2010-01-04       NaN       NaN
         2010-01-05  0.003111 -0.035038
         2010-01-06  0.000545 -0.009868
         2010-01-07  0.003993 -0.005233
         2010-01-08  0.002878 -0.050024

In [51]: rets.dropna(inplace=True)

In [52]: rets.plot(subplots=True, figsize=(10, 6));
```



图 8-10　一段时间内标普 500 与 VIX 的对数回报率

在这种情况下，pandas `scatter_matrix()`绘图函数对可视化来说很方便。该函数可绘制两个序列对数回报率的对比，用户可以在对角线上添加直方图或者核密度估计量（KDE）（见图 8-11）：

```
In [53]: pd.plotting.scatter_matrix(rets, ❶
                                    alpha=0.2, ❷
```

```
                                diagonal='hist', ❸
                                hist_kwds={'bins': 35}, ❹
                                figsize=(10, 6));
```

❶ 绘图所用的数据集。

❷ `alpha` 参数指定点的透明度。

❸ 对角线上放置的图表：列数据的直方图。

❹ 传递给直方图绘制函数的关键字。

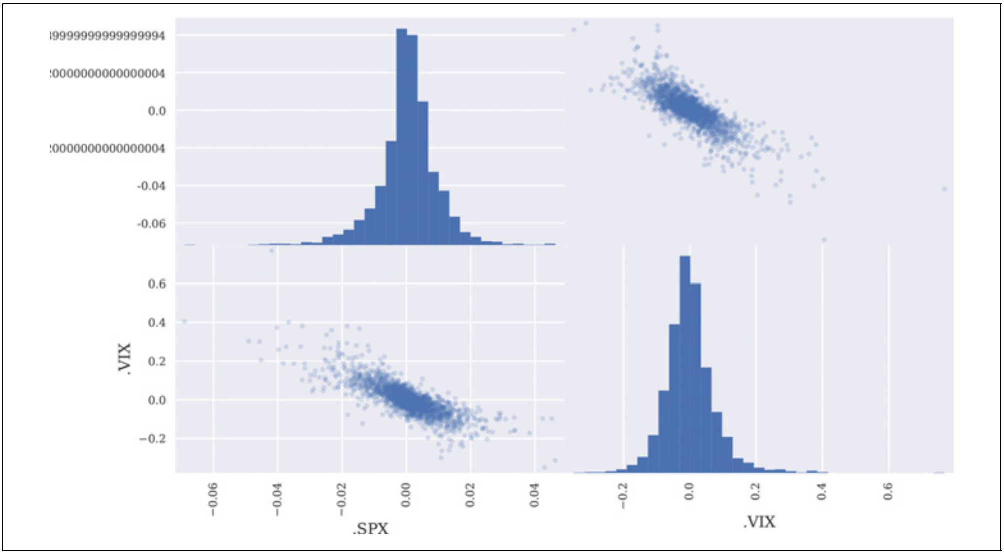


图 8-11　以散布矩阵形式表示的标普 500 和 VIX 对数回报率

8.3.3　OLS 回归

做了以上这些准备之后，就可以方便地实施普通最小二乘（OLS）回归了。图 8-12 展示了对数回报率的散点图，以及穿过“点云”的线性回归线。这条直线的斜率明显为负，为两个指数负相关的典型事实提供了支持：

```
In [54]: reg = np.polyfit(rets['.SPX'], rets['.VIX'], deg=1) ❶

In [55]: ax = rets.plot(kind='scatter', x='.SPX', y='.VIX', figsize=(10, 6)) ❷
         ax.plot(rets['.SPX'], np.polyval(reg, rets['.SPX']), 'r', lw=2); ❸
```

❶ 实施线性 OLS 回归。

❷ 绘制对数回报率散点图。

❸ 添加线性回归线。

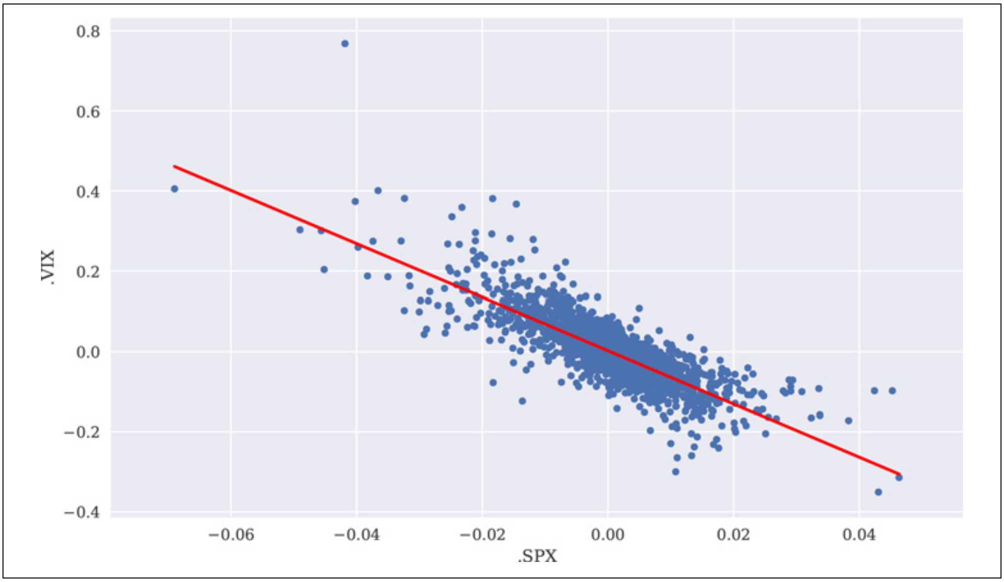


图 8-12　以散布矩阵形式表示的标普 500 和 VIX 对数回报率

8.3.4　相关

最后，我们直接考虑相关的度量。我们要考虑的度量有两种：静态度量考虑整个数据集，滚动度量显示一段时间内固定窗口上的相关。从图 8-13 可以看出，相关度确实随时间而改变，但是在给定参数下，它始终为负数。这为标普 500 与 VIX 指数（强）负相关的典型事实提供了有力支持：



图 8-13　标普 500 与 VIX 之间的相关度（静态和滚动）

```
In [56]: rets.corr() ❶
Out[56]:           .SPX      .VIX
         .SPX  1.000000 -0.804382
         .VIX -0.804382  1.000000

In [57]: ax = rets['.SPX'].rolling(window=252).corr(
                           rets['.VIX']).plot(figsize=(10, 6)) ❷
         ax.axhline(rets.corr().iloc[0, 1], c='r'); ❸
```

❶ 整个 `DataFrame` 的相关矩阵。

❷ 绘制一段时间的滚动相关度。

❸ 以水平线的形式在图表上添加静态值。

8.4 高频数据

本章介绍的是使用 pandas 进行的金融时间序列分析。分时数据集是金融时间序列的特例。老实说，它们可以或多或少地用相同方式处理，例如，本章迄今为止使用的日终数据集。用 pandas 导入这种数据集通常也相当快。这个数据集共有 17 352 行数据（参见图 8-14）：

```
In [59]: %%time
         # data from FXCM Forex Capital Markets Ltd.
         tick = pd.read_csv('../../source/fxcm_eur_usd_tick_data.csv',
                             index_col=0, parse_dates=True)
         CPU times: user 1.07 s, sys: 149 ms, total: 1.22 s
         Wall time: 1.16 s

In [60]: tick.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 461357 entries, 2018-06-29 00:00:00.082000 to 2018-06-29
          20:59:00.607000
         Data columns (total 2 columns):
         Bid    461357 non-null float64
         Ask    461357 non-null float64
         dtypes: float64(2)
         memory usage: 10.6 MB

In [61]: tick['Mid'] = tick.mean(axis=1) ❶

In [62]: tick['Mid'].plot(figsize=(10, 6));
```

❶ 计算每个数据行的中间价。

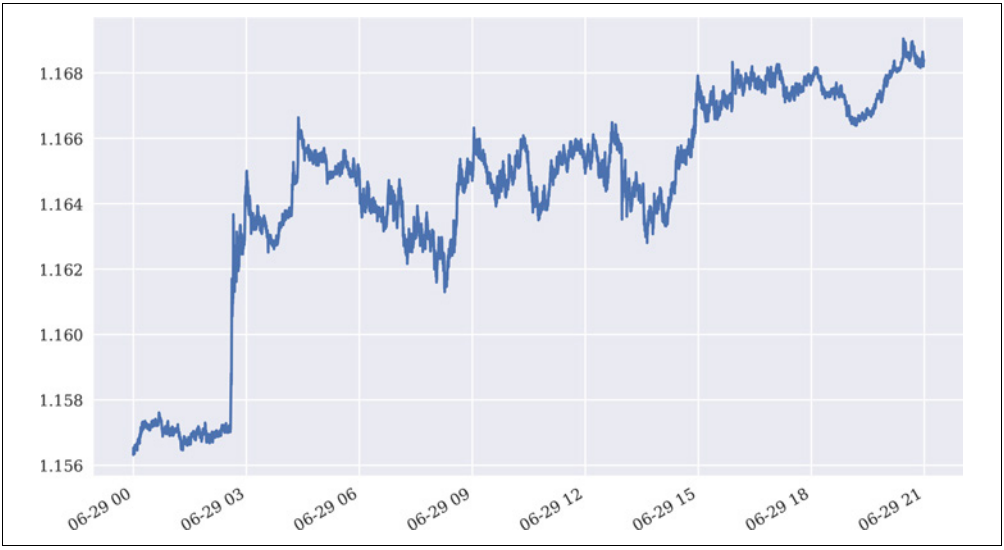


图 8-14　欧元/美元汇率分时数据

使用分时数据通常需要对金融时间序列进行重新采样。下面的代码重新采样分时数据，得到 5 分钟线数据（参见图 8-15）。这种数据此后可用于检验算法交易策略或者技术分析：

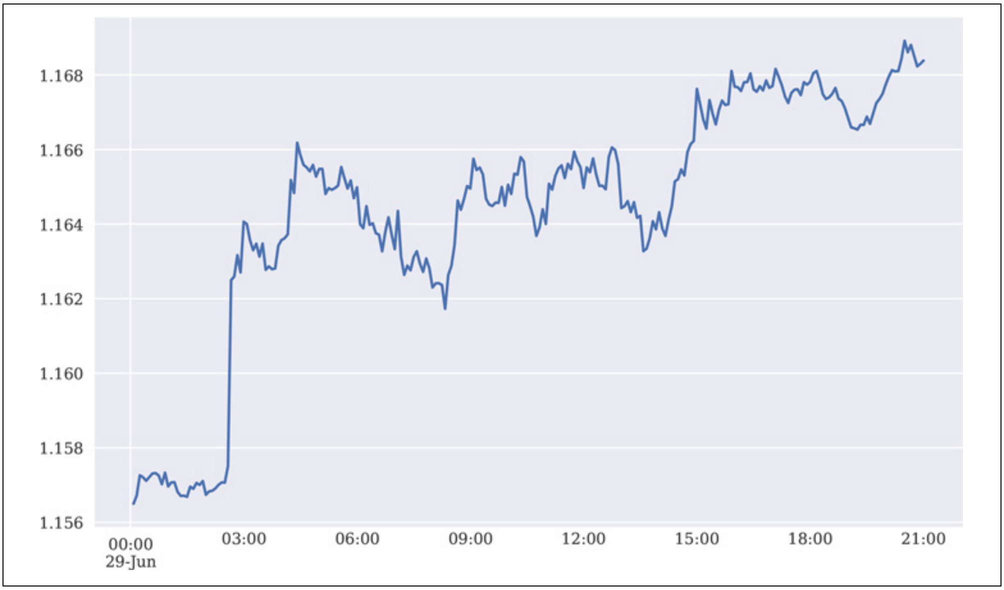


图 8-15　欧元/美元汇率的 5 分钟线数据

```
In [63]: tick_resam = tick.resample(rule='5min', label='right').last()

In [64]: tick_resam.head()
```

```
Out[64]:                              Bid      Ask       Mid
         2018-06-29 00:05:00 1.15649 1.15651 1.156500
         2018-06-29 00:10:00 1.15671 1.15672 1.156715
         2018-06-29 00:15:00 1.15725 1.15727 1.157260
         2018-06-29 00:20:00 1.15720 1.15722 1.157210
         2018-06-29 00:25:00 1.15711 1.15712 1.157115

In [65]: tick_resam['Mid'].plot(figsize=(10, 6));
```

8.5 结语

本章处理了金融时间序列，这是金融领域最为重要的数据类型之一。pandas 是处理这种数据集的强大程序库，不仅可以进行高效的数据分析，还有很多其他功能，如简易的可视化。pandas 也可以从不同来源读取此类数据集，以及将数据集导出为不同技术文件格式，后续章节将对此进行介绍。

8.6 延伸阅读

本章所涉主题的出色参考书如下：

- McKinney, Wes (2017). *Python for Data Analysis*. Sebastopol, CA: O'Reilly.
- VanderPlas, Jake (2016). *Python Data Science Handbook*. Sebastopol, CA: O'Reilly.

第 9 章

输入/输出操作

在未掌握数据之前就做出推测，是最大的错误。

——夏洛克•福尔摩斯

通用的原则是，不管是在金融环境还是其他任何应用领域，大部分数据都保存在硬盘驱动器（HDD）或者某种形式的固定存储设备上（如固态硬盘 SSD 或者混合型磁盘驱动器）。多年以来，设备的存储容量稳步增长，且每个存储单位（例如兆字节）的成本不断下降。

与此同时，存储数据量的增长速度远远快于典型的随机访问存储器（RAM）的增长速度，即使在最大型的机器上也是如此。这就使得，将数据存储到磁盘不仅可以持久存储，也可以通过将数据从 RAM 交换到磁盘来弥补 RAM 不足的问题。

输入/输出（I/O）操作通常是金融应用和数据密集型应用当中非常重要的任务。它们往往代表着性能关键计算的瓶颈，因为 I/O 操作一般无法以足够快的速度将数据写入 RAM[1] 和从 RAM 写入磁盘。在某种意义上，CPU 常常因为缓慢的 I/O 操作而“挨饿”。

尽管当今的大部分金融和企业分析工作面对的是“大数据”（例如 PB 级别），但单一分析任务使用的数据（子）集通常应该归入“中”数据的类别。微软研究部门的一项研究得出了以下结论：

> 我们的测算和最近的其他工作表明，现实世界中大部分分析工作处理的输入小于 100GB，但是 Hadoop/MapReduce 等流行基础架构最初是为 PB 级处理设计的。
>
> ——Appuswamy 等人（2013）

至于频率，单一金融分析任务通常处理的数据不超过几个 GB——这是 Python 及其科

1 在此，我们没有区分不同级别的 RAM 和处理器缓存。当前内存架构的最优使用方法本身就是一个主题。——原注

学栈中的库（如 NumPy、pandas 和 PyTables）最有效的处理能力。这样大小的数据集也可以在内存中分析，利用现在的 CPU 和 GPU 通常可以获得很高的速度。然而，数据必须读入 RAM、结果要写入磁盘，同时要确保满足当今的性能要求。

本章介绍如下内容。

Python 基本 I/O

Python 有内建的函数可以进行序列化对象、将内容存储到磁盘、从磁盘将内容读入 RAM 等操作。除此之外，Python 很擅长处理文本文件和 SQL 数据库。NumPy 也提供了专用于快速存储和检索 ndarray 对象的函数。

pandas 的 I/O

pandas 库提供了丰富的便利函数及方法，读取以不同格式存储的数据（例如 CSV、JSON），并将数据写入不同格式的文件中。

PyTables 的 I/O

PyTables 使用 HDF5 标准实现了大数据集的快速 I/O 操作；其速度往往只受到所使用硬件的限制。

TsTables 的 I/O

TsTables 是一个在 PyTables 基础上开发的软件包，可以快速存储和检索时间序列数据。

9.1 Python 基本 I/O

Python 本身自带多种 I/O 功能，有些为性能进行了优化，有些则更注重灵活性。不过，这些功能既可以方便地交互性使用，也可以在生产环境下使用。

9.1.1 将对象写入磁盘

为了以后使用文档或者与其他人共享文档，人们可能想将 Python 对象存储到磁盘上。办法之一是使用 pickle 模块。这个模块可以序列化大部分 Python 对象。序列化指的是将对象（层次结构）转换为一个字节流；反序列化是相反的操作。

和之前一样，首先导入一些与绘图有关的程序库和定制设置：

```
In [1]: from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

在下面的例子中，我们使用（伪）随机数据，这次将数据保存在一个列表对象中：

```
In [2]: import pickle ❶
        import numpy as np
        from random import gauss ❷

In [3]: a = [gauss(1.5, 2) for i in range(1000000)] ❸

In [4]: path = '/Users/yves/Temp/data/' ❹

In [5]: pkl_file = open(path + 'data.pkl', 'wb') ❺
```

❶ 从标准库中导入 `pickle` 模块。

❷ 导入 `gauss` 以生成正态分布随机数。

❸ 创建一个较大的列表对象，它包含随机数。

❹ 指定数据文件存储路径。

❺ 为二进制写入模式（`wb`）打开文件。

序列化和反序列化 Python 对象的两个重要函数是写入对象的 `dump()`和将对象加载到内存的 `load()`：

```
In [6]: %time pickle.dump(a, pkl_file) ❶
        CPU times: user 37.2 ms, sys: 15.3 ms, total: 52.5 ms
        Wall time: 50.8 ms

In [7]: pkl_file.close() ❷

In [8]: ll $path* ❸
        -rw-r--r-- 1 yves staff 9002006 Oct 19 12:11
        /Users/yves/Temp/data/data.pkl

In [9]: pkl_file = open(path + 'data.pkl', 'rb') ❹

In [10]: %time b = pickle.load(pkl_file) ❺
         CPU times: user 34.1 ms, sys: 16.7 ms, total: 50.8 ms
         Wall time: 48.7 ms

In [11]: a[:3]
Out[11]: [6.517874180585469, -0.5552400459507827, 2.8488946310833096]

In [12]: b[:3]
Out[12]: [6.517874180585469, -0.5552400459507827, 2.8488946310833096]

In [13]: np.allclose(np.array(a), np.array(b)) ❻
Out[13]: True
```

❶ 序列化对象 a，并将其保存到文件。

❷ 关闭文件。

❸ 显示磁盘文件及其大小（Mac/Linux）。

❹ 为二进制读模式（rb）打开文件。

❺ 从磁盘读取对象并反序列化。

❻ 将 a 和 b 转换为 ndarray 对象，np.allclose()验证两个对象是否包含相同的数据（数值）。

用 pickle 存储和读取单个对象相当简单，那么，读取两个对象呢？

```
In [14]: pkl_file = open(path + 'data.pkl', 'wb')

In [15]: %time pickle.dump(np.array(a), pkl_file) ❶
         CPU times: user 58.1 ms, sys: 6.09 ms, total: 64.2 ms
         Wall time: 32.5 ms

In [16]: %time pickle.dump(np.array(a) ** 2, pkl_file) ❷
         CPU times: user 66.7 ms, sys: 7.22 ms, total: 73.9 ms
         Wall time: 39.3 ms

In [17]: pkl_file.close()

In [18]: ll $path* ❸
         -rw-r--r-- 1 yves staff 16000322 Oct 19 12:11
         /Users/yves/Temp/data/data.pkl
```

❶ 序列化 a 的 ndarray 版本并保存。

❷ 序列化 a 平方数的 ndarray 版本并保存。

❸ 文件现在的大小大致为此前的两倍。

将两个 ndarray 对象读回内存怎么样？

```
In [19]: pkl_file = open(path + 'data.pkl', 'rb')

In [20]: x = pickle.load(pkl_file) ❶
         x[:4]
Out[20]: array([ 6.51787418, -0.55524005, 2.84889463, 5.94489175])

In [21]: y = pickle.load(pkl_file) ❷
         y[:4]
Out[21]: array([42.48268383, 0.30829151, 8.11620062, 35.34173791])

In [22]: pkl_file.close()
```

❶ 检索保存的第一个对象。

❷ 检索保存的第二个对象。

显然，`pickle` 按照先进先出（FIFO）原则保存对象。这种方法有一个重大问题：没有任何可用的元信息可以让用户事先知道保存在 `pickle` 文件中的是什么。

有时候可以采用一种变通的方法：不存储单个对象，而是存储包含所有其他对象的字典（dict）对象：

```
In [23]: pkl_file = open(path + 'data.pkl', 'wb')
         pickle.dump({'x': x, 'y': y}, pkl_file) ❶
         pkl_file.close()

In [24]: pkl_file = open(path + 'data.pkl', 'rb')
         data = pickle.load(pkl_file) ❷
         pkl_file.close()
         for key in data.keys():
             print(key, data[key][:4])
         x [ 6.51787418 -0.55524005 2.84889463 5.94489175]
         y [42.48268383 0.30829151 8.11620062 35.34173791]

In [25]: !rm -f $path*
```

❶ 保存包含两个 `ndarray` 对象的字典对象。

❷ 读取字典对象。

这种方法要求我们一次写入和读取所有对象，在许多情况下，人们可能需要为了更便利而容忍这个问题。

兼容性问题

使用 `pickle` 进行对象序列化通常很简单。但是，它可能造成一些问题，例如在某个 Python 程序库升级，或新版本无法使用旧版本的序列化对象时。不同平台和操作系统之间共享这些对象也可能带来问题。因此，通常建议使用 NumPy 和 pandas 等程序库内置的读取和写入功能，这将在后面几节里讨论。

9.1.2　读取和写入文本文件

文本处理是 Python 的优势之一。实际上，许多企业和科学用户就将 Python 用于这项任务。利用 Python 处理字符串对象和文本文件有许多选项。

假定我们已经生成了一组相当大的数据，打算将其保存为逗号分隔值（CSV）文件并

共享。尽管 CSV 文件有特殊的结构，但是这种文件本质上还是普通文本文件。下面的代码创建了一个 `ndarray` 对象形式的模拟数据集，还创建一个 `DatetimeIndex` 对象，并将两者组合起来保存为 CSV 文本文件：

```
In [26]: import pandas as pd

In [27]: rows = 5000 ❶
a = np.random.standard_normal((rows, 5)).round(4) ❷

In [28]: a ❷
Out[28]: array([[-0.0892, -1.0508, -0.5942, 0.3367, 1.508 ],
                [ 2.1046, 3.2623, 0.704 , -0.2651, 0.4461],
                [-0.0482, -0.9221, 0.1332, 0.1192, 0.7782],
                ...,
                [ 0.3026, -0.2005, -0.9947, 1.0203, -0.6578],
                [-0.7031, -0.6989, -0.8031, -0.4271, 1.9963],
                [ 2.4573, 2.2151, 0.158 , -0.7039, -1.0337]])

In [29]: t = pd.date_range(start='2019/1/1', periods=rows, freq='H') ❸

In [30]: t ❸
Out[30]: DatetimeIndex(['2019-01-01 00:00:00', '2019-01-01 01:00:00',
                        '2019-01-01 02:00:00', '2019-01-01 03:00:00',
                        '2019-01-01 04:00:00', '2019-01-01 05:00:00',
                        '2019-01-01 06:00:00', '2019-01-01 07:00:00',
                        '2019-01-01 08:00:00', '2019-01-01 09:00:00',
                        ...
                        '2019-07-27 22:00:00', '2019-07-27 23:00:00',
                        '2019-07-28 00:00:00', '2019-07-28 01:00:00',
                        '2019-07-28 02:00:00', '2019-07-28 03:00:00',
                        '2019-07-28 04:00:00', '2019-07-28 05:00:00',
                        '2019-07-28 06:00:00', '2019-07-28 07:00:00'],
                        dtype='datetime64[ns]', length=5000, freq='H')

In [31]: csv_file = open(path + 'data.csv', 'w') ❹

In [32]: header = 'date,no1,no2,no3,no4,no5\n' ❺

In [33]: csv_file.write(header) ❺
Out[33]: 25

In [34]: for t_, (no1, no2, no3, no4, no5) in zip(t, a): ❻
             s = '{},{},{},{},{},{}\n'.format(t_, no1, no2, no3, no4, no5) ❼
             csv_file.write(s) ❽

In [35]: csv_file.close()
```

```
In [36]: ll $path*
         -rw-r--r-- 1 yves staff 284757 Oct 19 12:11
          /Users/yves/Temp/data/data.csv
```

❶ 定义数据集行数。

❷ 创建包含随机数的 `ndarray` 对象。

❸ 创建长度合适（间隔为一小时）的 `DatetimeIndex` 对象。

❹ 打开文件以便写入。

❺ 定义标题行（列标签），并将其作为第一行写入。

❻ 按行合并数据。

❼ 字符串对象。

❽ 逐行写入（附加到 CSV 文本文件）。

另一种方法与上述方法类似。首先，打开现存的 CSV 文件。其次，用文件对象的 `readline()`或 `readlines()`方法逐行读入其内容：

```
In [37]: csv_file = open(path + 'data.csv', 'r') ❶

In [38]: for i in range(5):
             print(csv_file.readline(), end='') ❷
         date,no1,no2,no3,no4,no5
         2019-01-01 00:00:00,-0.0892,-1.0508,-0.5942,0.3367,1.508
         2019-01-01 01:00:00,2.1046,3.2623,0.704,-0.2651,0.4461
         2019-01-01 02:00:00,-0.0482,-0.9221,0.1332,0.1192,0.7782
         2019-01-01 03:00:00,-0.359,-2.4955,0.6164,0.712,-1.4328

In [39]: csv_file.close()

In [40]: csv_file = open(path + 'data.csv', 'r') ❶

In [41]: content = csv_file.readlines() ❸

In [42]: content[:5] ❹
Out[42]: ['date,no1,no2,no3,no4,no5\n',
          '2019-01-01 00:00:00,-0.0892,-1.0508,-0.5942,0.3367,1.508\n',
          '2019-01-01 01:00:00,2.1046,3.2623,0.704,-0.2651,0.4461\n',
          '2019-01-01 02:00:00,-0.0482,-0.9221,0.1332,0.1192,0.7782\n',
          '2019-01-01 03:00:00,-0.359,-2.4955,0.6164,0.712,-1.4328\n']

In [43]: csv_file.close()
```

❶ 打开文件以读取（`r`）。

❷ 逐行读取文件内容并打印。

❸ 用一个步骤读取文件内容。

❹ 结果是一个列表对象，包含表示所有行的单独字符串对象。

CSV 文件很重要也很常见，因此 Python 标准库中有一个 csv 模块，它可以简化这些文件的处理。csv 模块有两个很有益的读取器（迭代器）对象，可返回列表对象或者字典对象的一个列表：

```
In [44]: import csv

In [45]: with open(path + 'data.csv', 'r') as f:
             csv_reader = csv.reader(f) ❶
             lines = [line for line in csv_reader]

In [46]: lines[:5]❶
Out[46]: [['date', 'no1', 'no2', 'no3', 'no4', 'no5'],
         ['2019-01-01 00:00:00', '-0.0892', '-1.0508', '-0.5942', '0.3367',
          '1.508'],
         ['2019-01-01 01:00:00', '2.1046', '3.2623', '0.704', '-0.2651',
          '0.4461'],
         ['2019-01-01 02:00:00', '-0.0482', '-0.9221', '0.1332', '0.1192',
          '0.7782'],
         ['2019-01-01 03:00:00', '-0.359', '-2.4955', '0.6164', '0.712',
          '-1.4328']]

In [47]: with open(path + 'data.csv', 'r') as f:
              csv_reader = csv.DictReader(f) ❷
              lines = [line for line in csv_reader]

In [48]: lines[:3] ❷
Out[48]: [OrderedDict([('date', '2019-01-01 00:00:00'),
                       ('no1', '-0.0892'),
                       ('no2', '-1.0508'),
                       ('no3', '-0.5942'),
                       ('no4', '0.3367'),
                       ('no5', '1.508')]),
         OrderedDict([('date', '2019-01-01 01:00:00'),
                       ('no1', '2.1046'),
                       ('no2', '3.2623'),
                       ('no3', '0.704'),
                       ('no4', '-0.2651'),
                       ('no5', '0.4461')]),
         OrderedDict([('date', '2019-01-01 02:00:00'),
                       ('no1', '-0.0482'),
```

```
                              ('no2', '-0.9221'),
                              ('no3', '0.1332'),
                              ('no4', '0.1192'),
                              ('no5', '0.7782')])]

In [49]: !rm -f $path*
```

❶ `csv.reader()`将每一行当成一个列表对象返回。

❷ `csv.DictReader()`将每一行当成 `OrderedDict`（有序字典）对象返回，该对象是字典对象的特例。

9.1.3　使用 SQL 数据库

Python 可以使用任何类型的结构化查询语言（SQL）数据库，通常也能处理任何 NoSQL 数据库。默认随 Python 一同交付的 SQL 数据库（关系数据库）是 SQLite3。可以利用它来说明 Python 处理 SQL 数据库的方法：[1]

```
In [50]: import sqlite3 as sq3

In [51]: con = sq3.connect(path + 'numbs.db') ❶

In [52]: query = 'CREATE TABLE numbs (Date date, No1 real, No2 real)' ❷

In [53]: con.execute(query) ❸
Out[53]: <sqlite3.Cursor at 0x102655f10>

In [54]: con.commit() ❹

In [55]: q = con.execute ❺

In [56]: q('SELECT * FROM sqlite_master').fetchall() ❻
Out[56]: [('table',
           'numbs',
           'numbs',
           2,
           'CREATE TABLE numbs (Date date, No1 real, No2 real)')]
```

❶ 打开一个数据库连接；创建一个文件（如果不存在）。

❷ 这个 SQL 查询创建一个包含 3 列的表。

❸ 执行查询。

1　Python 可用数据库连接器的概述详见 Python 官方文档。事实证明，使用 SQLALchemy 等对象关系映射器往往比直接访问关系数据库更实用。这些映射器引入一个抽象层，可以实现更具 Python 风格的面向对象代码，而且，它们还可以帮助你更轻松地更换后端的关系数据库。——原注

❹ 提交更改。

❺ 为 `con.execute()` 方法定义一个短别名。

❻ 读取关于数据库的元信息，将刚刚创建的表显示为单个对象。

现在，我们有了一个包含单个表的数据库，可以在表中填入数据了。每行数据包括一个日期时间（datetime）对象和两个浮点（float）对象：

```
In [57]: import datetime

In [58]: now = datetime.datetime.now()
         q('INSERT INTO numbs VALUES(?, ?, ?)', (now, 0.12, 7.3)) ❶
Out[58]: <sqlite3.Cursor at 0x102655f80>

In [59]: np.random.seed(100)

In [60]: data = np.random.standard_normal((10000, 2)).round(4) ❷

In [61]: %%time
         for row in data: ❸
             now = datetime.datetime.now()
             q('INSERT INTO numbs VALUES(?, ?, ?)', (now, row[0], row[1]))
         con.commit()
         CPU times: user 115 ms, sys: 6.69 ms, total: 121 ms
         Wall time: 124 ms

In [62]: q('SELECT * FROM numbs').fetchmany(4) ❹
Out[62]: [('2018-10-19 12:11:15.564019', 0.12, 7.3),
          ('2018-10-19 12:11:15.592956', -1.7498, 0.3427),
          ('2018-10-19 12:11:15.593033', 1.153, -0.2524),
          ('2018-10-19 12:11:15.593051', 0.9813, 0.5142)]

In [63]: q('SELECT * FROM numbs WHERE no1 > 0.5').fetchmany(4) ❺
Out[63]: [('2018-10-19 12:11:15.593033', 1.153, -0.2524),
          ('2018-10-19 12:11:15.593051', 0.9813, 0.5142),
          ('2018-10-19 12:11:15.593104', 0.6727, -0.1044),
          ('2018-10-19 12:11:15.593134', 1.619, 1.5416)]

In [64]: pointer = q('SELECT * FROM numbs') ❻

In [65]: for i in range(3):
             print(pointer.fetchone()) ❼
         ('2018-10-19 12:11:15.564019', 0.12, 7.3)
         ('2018-10-19 12:11:15.592956', -1.7498, 0.3427)
         ('2018-10-19 12:11:15.593033', 1.153, -0.2524)
```

```
In [66]: rows = pointer.fetchall() ❽
         rows[:3]
Out[66]: [('2018-10-19 12:11:15.593051', 0.9813, 0.5142),
          ('2018-10-19 12:11:15.593063', 0.2212, -1.07),
          ('2018-10-19 12:11:15.593073', -0.1895, 0.255)]
```

❶ 写入一行（记录）到 `numbs` 表。

❷ 以 `ndarray` 对象形式创建一个较大的虚拟数据集。

❸ 循环读取 `ndarray` 对象各行。

❹ 从表中读取一些行。

❺ 同上，但是以 No1 列上的值为条件。

❻ 定义一个指针对象。

❼ 表现类似于生成器对象。

❽ 读取剩下的所有行。

最后，你可能想要在不需要数据库中的表对象时将其删除：

```
In [67]: q('DROP TABLE IF EXISTS numbs') ❶
Out[67]: <sqlite3.Cursor at 0x1187a7420>

In [68]: q('SELECT * FROM sqlite_master').fetchall() ❷
Out[68]: []

In [69]: con.close() ❸

In [70]: !rm -f $path* ❹
```

❶ 从数据库中删除表。

❷ 操作之后没有任何表对象。

❸ 关闭数据库连接。

❹ 从磁盘上删除数据库文件。

SQL 数据库是相当广泛的主题，本章无法涵盖其广度和复杂度，只能提供一些基本信息：

- Python 和几乎所有数据库技术都能很好地集成；
- 基本 SQL 语法主要由使用的数据库决定，剩下的都是真正的 Python 风格。

本章后面还有几个基于 SQLite3 的例子。

9.1.4　读写 NumPy 数组

NumPy 本身有以便利、高性能的方式写入和读取 `ndarray` 对象的函数。这在某些情况下节省了工作量，例如必须将 NumPy `dtype` 转换为特定数据库类型（例如 SQLite3）时。为了说明 NumPy 有时候是基于 SQL 方法的高效替代品，我们用 NumPy 重复前面的例子。下列代码中用 NumPy 的 `np.arrange()` 函数代替 `pandas`，生成存储 `datetime` 对象的 `ndarray` 对象：

```
In [71]: dtimes = np.arange('2019-01-01 10:00:00', '2025-12-31 22:00:00',
                            dtype='datetime64[m]') ❶

In [72]: len(dtimes)
Out[72]: 3681360

In [73]: dty = np.dtype([('Date', 'datetime64[m]'),
                         ('No1', 'f'), ('No2', 'f')]) ❷

In [74]: data = np.zeros(len(dtimes), dtype=dty) ❸

In [75]: data['Date'] = dtimes ❹

In [76]: a = np.random.standard_normal((len(dtimes), 2)).round(4) ❺

In [77]: data['No1'] = a[:, 0] ❻
         data['No2'] = a[:, 1] ❻

In [78]: data.nbytes ❼
Out[78]: 58901760
```

❶ 创建一个 `ndarray` 对象，将其 `dtype` 设置为 `datetime`。

❷ 为结构数组定义特殊的 `dtype` 对象。

❸ 用特殊 `dtype` 来实例化 `ndarray` 对象。

❹ 填充 `Date` 列。

❺ 模拟数据集。

❻ 填入 `No1` 和 `No2` 列。

❼ 结构数组的大小，以字节表示。

`ndarray` 对象的保存经过了高度优化，因此相当快速。在磁盘（这里使用 SSD）中保存近 60MB 的数据只花了不到 0.1s。包含 480MB 数据的较大 `ndarray` 对象保存在磁盘上也只花了大约 0.5s：[1]

1　注意，这一时间即使在同一台机器上多次重复也可能有显著的不同，因为除了其他因素之外，它们还取决于此时机器的 CPU 和 I/O 系统在执行什么任务。——原注

```
In [79]: %time np.save(path + 'array', data) ❶
         CPU times: user 37.4 ms, sys: 58.9 ms, total: 96.4 ms
         Wall time: 77.9 ms

In [80]: ll $path* ❷
         -rw-r--r-- 1 yves staff 58901888 Oct 19 12:11
         /Users/yves/Temp/data/array.npy

In [81]: %time np.load(path + 'array.npy') ❸
         CPU times: user 1.67 ms, sys: 44.8 ms, total: 46.5 ms
         Wall time: 44.6 ms

Out[81]: array([('2019-01-01T10:00', 1.5131, 0.6973),
                ('2019-01-01T10:01', -1.722 , -0.4815),
                ('2019-01-01T10:02', 0.8251, 0.3019), ...,
                ('2025-12-31T21:57', 1.372 , 0.6446),
                ('2025-12-31T21:58', -1.2542, 0.1612),
                ('2025-12-31T21:59', -1.1997, -1.097 )],
                dtype=[('Date', '<M8[m]'), ('No1', '<f4'), ('No2', '<f4')])

In [82]: %time data = np.random.standard_normal((10000, 6000)).round(4) ❹
         CPU times: user 2.69 s, sys: 391 ms, total: 3.08 s
         Wall time: 2.78 s

In [83]: data.nbytes ❹
Out[83]: 480000000

In [84]: %time np.save(path + 'array', data) ❹
         CPU times: user 42.9 ms, sys: 300 ms, total: 343 ms
         Wall time: 481 ms

In [85]: ll $path*❹
         -rw-r--r-- 1 yves staff 480000128 Oct 19 12:11
         /Users/yves/Temp/data/array.npy

In [86]: %time np.load(path + 'array.npy')❹
         CPU times: user 2.32 ms, sys: 363 ms, total: 365 ms
         Wall time: 363 ms

Out[86]: array([[ 0.3066, 0.5951, 0.5826, ..., 1.6773, 0.4294, -0.2216],
                [ 0.8769, 0.7292, -0.9557, ..., 0.5084, 0.9635, -0.4443],
                [-1.2202, -2.5509, -0.0575, ..., -1.6128, 0.4662, -1.3645],
                ...,
                [-0.5598, 0.2393, -2.3716, ..., 1.7669, 0.2462, 1.035 ],
                [ 0.273 , 0.8216, -0.0749, ..., -0.0552, -0.8396, 0.3077],
                [-0.6305, 0.8331, 1.3702, ..., 0.3493, 0.1981, 0.2037]])

In [87]: !rm -f $path*
```

❶ 在磁盘上保存结构化 `ndarray` 对象。

❷ 磁盘上的尺寸几乎与内存中的相同（由于二进制存储）。

❸ 从磁盘加载结构化 `ndarray` 对象。

❹ 更大的常规 `ndarray` 对象。

这些例子说明，写入磁盘主要受到硬件的限制，因为观察到的速度大约相当于本书编著时较好的 SSD 广告中声称的写入速度（512MB/s）。

在任何情况下我们都可以预期，这种形式的数据存储和检索远快于 SQL 数据库或者使用 `pickle` 模块的序列化。原因有二：首先，数据主要是数值；其次，NumPy 使用二进制存储，几乎将额外开销下降到 0。当然，这种方法没有 SQL 数据库的功能性，但是后面内容中介绍的 PyTables 对此会有所帮助。

9.2 pandas 的 I/O

pandas 库的主要优势之一是可以原生读取和写入不同的数据格式，包括：

- CSV（逗号分隔值）;
- SQL（结构化查询语言）;
- XLS/XSLX（Microsoft Excel 文件）;
- JSON（JavaScript 对象标记法）;
- HTML（超文本标记语言）。

表9-1列出了所有的支持格式和对应的 `pandas` 及 `DataFrame` 类导入和导出的函数/方法。例如，导入函数（如 `pd.read_csv()`）使用的参数在 `pandas.read_csv` 文档中描述。

表 9-1　导入—导出函数及方法

格式	输入	输出	备注
CSV	`pd.read_csv()`	`.to_csv()`	文本文件
XLS/XLSX	`pd.read_excel()`	`.to_excel()`	电子表格
HDF	`pd.read_hdf()`	`.to_hdf()`	HDF5 数据库
SQL	`pd.read_sql()`	`.to_sql()`	SQL 表
JSON	`pd.read_json()`	`.to_json()`	JavaScript 对象标记
MSGPACK	`pd.read_msgpack()`	`.to_msgpack()`	可移植二进制格式
HTML	`pd.read_html()`	`.to_html()`	HTML 代码

续表

格式	输入	输出	备注
GBQ	`pd.read_gbq()`	`.to_gbq()`	Google Big Query 格式
DTA	`pd.read_stata()`	`.to_stata()`	格式 104、105、108、113～115、117
任意	`pd.read_clipboard()`	`.to_clipboard()`	例如，从 HTML 页面
任意	`pd.read_pickle()`	`.to_pickle()`	（结构化）Python 对象

我们的测试用例仍然是较大的一组浮点数：

```
In [88]: data = np.random.standard_normal((1000000, 5)).round(4)

In [89]: data[:3]
Out[89]: array([[ 0.4918, 1.3707, 0.137 , 0.3981, -1.0059],
                [ 0.4516, 1.4445, 0.0555, -0.0397, 0.44 ],
                [ 0.1629, -0.8473, -0.8223, -0.4621, -0.5137]])
```

为此，我们还将再次访问 SQLite3，比较使用 pandas 的替代方法的性能。

9.2.1 使用 SQL 数据库

现在，你应该对下面关于 SQLite3 的代码应该很熟悉了：

```
In [90]: filename = path + 'numbers'

In [91]: con = sq3.Connection(filename + '.db')

In [92]: query = 'CREATE TABLE numbers (No1 real, No2 real,\
                  No3 real, No4 real, No5 real)' ❶

In [93]: q = con.execute
         qm = con.executemany

In [94]: q(query)
Out[94]: <sqlite3.Cursor at 0x1187a76c0>
```

❶ 创建一个有 5 列的表以用于实数（浮点对象）。

这次可以应用`.executemany()`，因为数据在单个 `ndarray` 对象中。读取和使用数据的方法和以前的相同。查询结果也很容易可视化（见图 9-1）：

```
In [95]: %%time
         qm('INSERT INTO numbers VALUES (?, ?, ?, ?, ?)', data) ❶
         con.commit()
         CPU times: user 7.3 s, sys: 195 ms, total: 7.49 s
         Wall time: 7.71 s
```

```
In [96]: ll $path*
         -rw-r--r-- 1 yves staff 52633600 Oct 19 12:11
         /Users/yves/Temp/data/numbers.db

In [97]: %%time
         temp = q('SELECT * FROM numbers').fetchall() ❷
         print(temp[:3])
         [(0.4918, 1.3707, 0.137, 0.3981, -1.0059), (0.4516, 1.4445, 0.0555,
          -0.0397, 0.44), (0.1629, -0.8473, -0.8223, -0.4621, -0.5137)]
         CPU times: user 1.7 s, sys: 124 ms, total: 1.82 s
         Wall time: 1.9 s

In [98]: %%time
         query = 'SELECT * FROM numbers WHERE No1 > 0 AND No2 < 0'
         res = np.array(q(query).fetchall()).round(3) ❸
         CPU times: user 639 ms, sys: 64.7 ms, total: 704 ms
         Wall time: 702 ms

In [99]: res = res[::100] ❹
         plt.figure(figsize=(10, 6))
         plt.plot(res[:, 0], res[:, 1], 'ro') ❹
```

❶ 只用一步即可将整个数据集插入表。

❷ 只用一步即可从表中读取所有行。

❸ 有选择地读取一些行，将其转换成 `ndarray` 对象。

❹ 绘制查询结果中某个子集的图表。

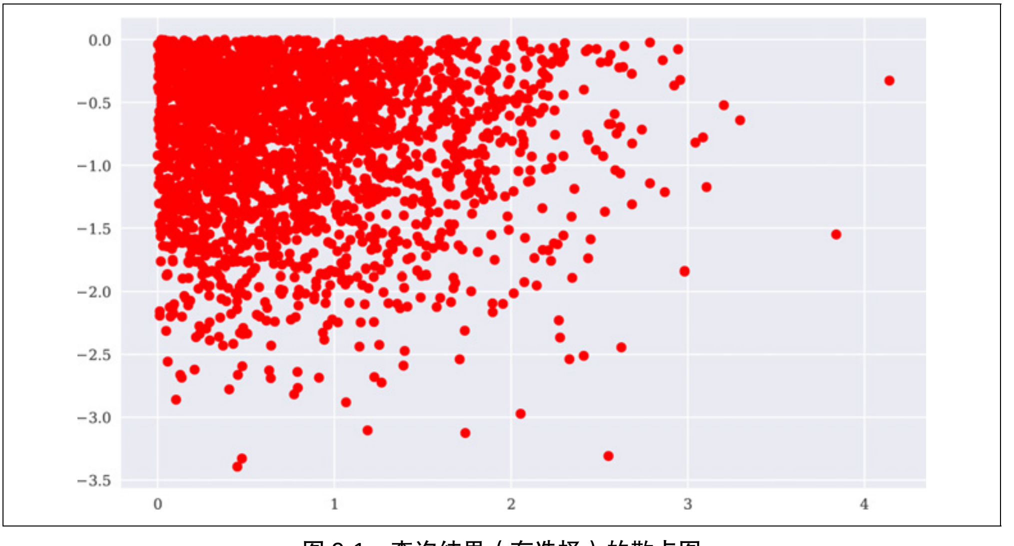


图 9-1　查询结果（有选择）的散点图

9.2.2　从 SQL 到 pandas

用 pandas 读取整个表或者查询结果通常更为高效。在可以将整个表读入内存时，分析查询的执行通常比使用基于磁盘的 SQL 方法快得多。

用 pandas 读取整个表花费的总时间和读入 NumPy 的 `ndarray` 对象的时间大致相同。瓶颈同样在 SQL 数据库：

```
In [100]: %time data = pd.read_sql('SELECT * FROM numbers', con) ❶
          CPU times: user 2.17 s, sys: 180 ms, total: 2.35 s
          Wall time: 2.32 s

In [101]: data.head()
Out[101]:       No1     No2     No3     No4     No5
          0  0.4918  1.3707  0.1370  0.3981 -1.0059
          1  0.4516  1.4445  0.0555 -0.0397  0.4400
          2  0.1629 -0.8473 -0.8223 -0.4621 -0.5137
          3  1.3064  0.9125  0.5142 -0.7868 -0.3398
          4 -0.1148 -1.5215 -0.7045 -1.0042 -0.0600
```

❶ 将表的所有行读入名为 `data` 的 `DataFrame` 对象中。

数据现在在内存中，可以对其进行更快的分析了，速度往往有一个或多个数量级的提升。pandas 还可以进行更为复杂的查询，但在复杂的关系数据结构下，这并不意味着可以取代 SQL 数据库。多重条件查询结果如图 9-2 所示：

```
In [102]: %time data[(data['No1'] > 0) & (data['No2'] < 0)].head() ❶
          CPU times: user 47.1 ms, sys: 12.3 ms, total: 59.4 ms
          Wall time: 33.4 ms

Out[102]:        No1     No2     No3     No4     No5
           2  0.1629 -0.8473 -0.8223 -0.4621 -0.5137
           5  0.1893 -0.0207 -0.2104  0.9419  0.2551
           8  1.4784 -0.3333 -0.7050  0.3586 -0.3937
          10  0.8092 -0.9899  1.0364 -1.0453  0.0579
          11  0.9065 -0.7757 -0.9267  0.7797  0.0863

In [103]: %%time
          q = '(No1 < -0.5 | No1 > 0.5) & (No2 < -1 | No2 > 1)' ❷
          res = data[['No1', 'No2']].query(q) ❷
          CPU times: user 95.4 ms, sys: 22.4 ms, total: 118 ms
          Wall time: 56.4 ms

In [104]: plt.figure(figsize=(10, 6))
          plt.plot(res['No1'], res['No2'], 'ro');
```

❶ 2 个逻辑条件组合。

❷ 4 个逻辑条件组合。

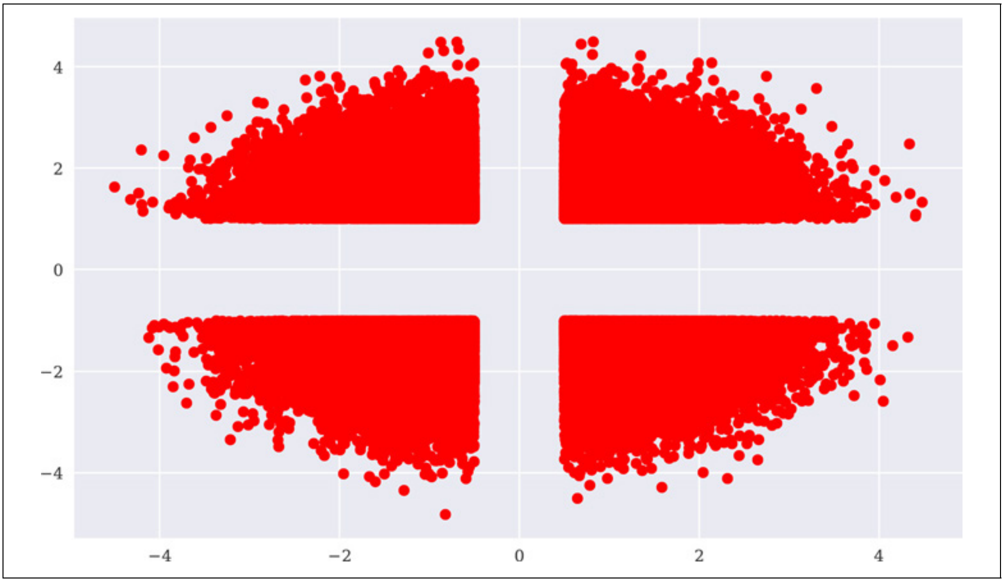


图 9-2　查询结果（有选择）散点图

正如你所预期的，只要 pandas 能够复制对应的 SQL 语句，那么使用 pandas 的内存分析能力可以有效提升速度。这并不是使用 pandas 的唯一优势，因为 pandas 与其他程序库（包括下一节的主题 PyTables）紧密集成。在此，知道这种组合能够大大加速 I/O 操作就足够了。下面的例子就能说明这一点：

```
In [105]: h5s = pd.HDFStore(filename + '.h5s', 'w') ❶

In [106]: %time h5s['data'] = data ❷
          CPU times: user 46.7 ms, sys: 47.1 ms, total: 93.8 ms
          Wall time: 99.7 ms

In [107]: h5s ❸
Out[107]: <class 'pandas.io.pytables.HDFStore'>
          File path: /Users/yves/Temp/data/numbers.h5s

In [108]: h5s.close() ❹
```

❶ 打开一个 HDF5 数据库文件用于写入；在 pandas 中，创建一个 `HDFStore` 对象。

❷ 完整的 `DataFrame` 对象通过二进制存储被保存在数据库文件中。

❸ `HDFStore` 对象信息。

❹ 关闭数据库文件。

与 SQLite3 的相同操作相比，写入包含原始 SQL 表所有数据的整个 `DataFrame` 要快得多，读取速度也更快：

```
In [109]: %%time
          h5s = pd.HDFStore(filename + '.h5s', 'r') ❶
          data_ = h5s['data'] ❷
          h5s.close() ❸
          CPU times: user 11 ms, sys: 18.3 ms, total: 29.3 ms
          Wall time: 29.4 ms

In [110]: data_ is data ❹
Out[110]: False

In [111]: (data_ == data).all() ❺
Out[111]: No1    True
          No2    True
          No3    True
          No4    True
          No5    True
          dtype: bool
In [112]: np.allclose(data_, data) ❺
Out[112]: True

In [113]: ll $path* ❻
          -rw-r--r-- 1 yves staff 52633600 Oct 19 12:11
          /Users/yves/Temp/data/numbers.db
          -rw-r--r-- 1 yves staff 48007240 Oct 19 12:11
          /Users/yves/Temp/data/numbers.h5s
```

❶ 打开 HDF5 数据库以读取。

❷ 读取 `DataFrame`，将其保存在内存中，形式为 `data_`。

❸ 关闭数据库文件。

❹ 两个 `DataFrame` 对象不同。

❺ 包含相同数据。

❻ 二进制存储的尺寸通常比其他格式（例如 SQL 表）小。

9.2.3　使用 CSV 文件

CSV 格式是最广为使用的数据交换格式之一。虽然它没有真正标准化，但是可以由任何平台和大部分数据、金融分析应用程序处理。前面，我们已经了解了用标准 Python

功能写入和读取 CSV 文件中数据（参见 9.1.2 节）的方法。pandas 使这整个过程变得更加方便，代码更加简洁，执行通常也更快（参见图 9-3）：

```
In [114]: %time data.to_csv(filename + '.csv') ❶
          CPU times: user 6.44 s, sys: 139 ms, total: 6.58 s
          Wall time: 6.71 s

In [115]: ll $path
          total 283672
          -rw-r--r-- 1 yves staff 43834157 Oct 19 12:11 numbers.csv
          -rw-r--r-- 1 yves staff 52633600 Oct 19 12:11 numbers.db
          -rw-r--r-- 1 yves staff 48007240 Oct 19 12:11 numbers.h5s

In [116]: %time df = pd.read_csv(filename + '.csv') ❷
          CPU times: user 1.12 s, sys: 111 ms, total: 1.23 s
          Wall time: 1.23 s

In [117]: df[['No1', 'No2', 'No3', 'No4']].hist(bins=20, figsize=(10, 6));
```

❶ `.to_csv()`方法以 CSV 格式将 `DataFrame` 数据写入磁盘。

❷ `pd.read_csv()`方法将数据读回内存，成为新的 `DataFrame` 对象。

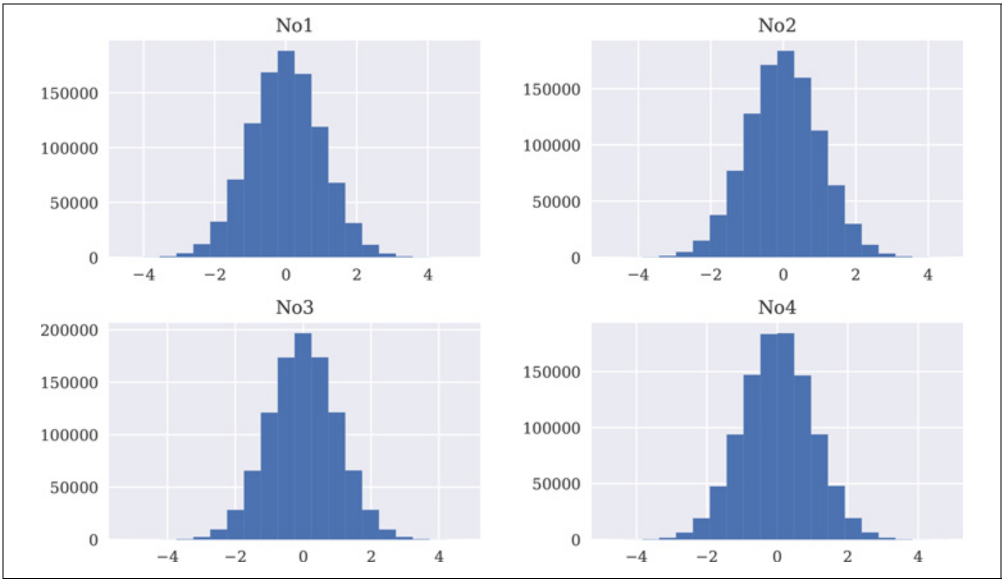


图 9-3　选择列的直方图

9.2.4　使用 Excel 文件

使用 Excel 数据表是后续章节的主题，现在我们只是简短地阐述 pandas 如何以 Excel

格式写入数据，以及如何从 Excel 数据表中读取数据。在该例中，我们将数据集限制在 10 万行（见图 9-4）：



图 9-4　所有列的线图

```
In [118]: %time data[:100000].to_excel(filename + '.xlsx') ❶
          CPU times: user 25.9 s, sys: 520 ms, total: 26.4 s
          Wall time: 27.3 s

In [119]: %time df = pd.read_excel(filename + '.xlsx', 'Sheet1') ❷
          CPU times: user 5.78 s, sys: 70.1 ms, total: 5.85 s
          Wall time: 5.91 s

In [120]: df.cumsum().plot(figsize=(10, 6));
In [121]: ll $path*
          -rw-r--r-- 1 yves staff 43834157 Oct 19 12:11
          /Users/yves/Temp/data/numbers.csv
          -rw-r--r-- 1 yves staff 52633600 Oct 19 12:11
          /Users/yves/Temp/data/numbers.db
          -rw-r--r-- 1 yves staff 48007240 Oct 19 12:11
          /Users/yves/Temp/data/numbers.h5s
          -rw-r--r-- 1 yves staff 4032725 Oct 19 12:12
          /Users/yves/Temp/data/numbers.xlsx

In [122]: rm -f $path*
```

❶ `.to_excel()`方法以 XLSX 格式将 `DataFrame` 数据写入磁盘。

❷ pd.read_excel()方法将数据读入内存，成为一个新的 DataFrame 对象，你也可以指定要读取的工作表。

用这个小的数据子集生成 Excel 数据表需要花费一段时间。这展现了数据表结构带来的开销。

对生成的文件进行检查后发现，DataFrame 与 HDFStore 相结合是最为紧凑的备选方法之一（使用本章后面介绍的压缩可以进一步加强这一优势）。同样的数据量如果用 CSV 文件（即文本文件）存储，文件尺寸会更大。这是使用 CSV 文件的性能较低的原因之一，另一个原因是它们“只是”一般的文本文件。

9.3 PyTables 的 I/O

PyTables 是 Python 与 HDF5 数据库/文件标准的结合。它专门为优化 I/O 操作的性能、最大限度地利用可用硬件而设计。该库的导入名称为 tables。在内存分析方面，PyTables 与 pandas 类似，并不是用于代替 SQL 数据库的，而是引入某些功能来进一步弥补不足。例如，PyTables 数据库可以有许多表，且它支持压缩和索引，以及表的重要查询。此外，它可以高效地存储 NumPy 数组，并具有自己独特的类数组数据结构。

首先导入程序库：

```
In [123]: import tables as tb ❶
          import datetime as dt
```

❶ 包名为 PyTables，导入名称为 tables。

9.3.1 使用表

PyTables 提供了基于文件的数据库格式，与 SQLite 类似[1]。下面的代码可以打开一个数据库文件，然后创建一个表：

```
In [124]: filename = path + 'pytab.h5'

In [125]: h5 = tb.open_file(filename, 'w') ❶

In [126]: row_des = {
              'Date': tb.StringCol(26, pos=1), ❷
              'No1': tb.IntCol(pos=2), ❸
              'No2': tb.IntCol(pos=3), ❸
```

1 其他数据需要服务器/客户端架构。事实证明，对于交互式数据与金融分析，基于文件的数据库更方便一些，也足以应付大部分场合。——原注

```
                 'No3': tb.Float64Col(pos=4), ❹
                 'No4': tb.Float64Col(pos=5) ❹
                 }
In [127]: rows = 2000000

In [128]: filters = tb.Filters(complevel=0) ❺

In [129]: tab = h5.create_table('/', 'ints_floats', ❻
                                row_des, ❼
                                title='Integers and Floats', ❽
                                expectedrows=rows, ❾
                                filters=filters) ❿

In [130]: type(tab)
Out[130]: tables.table.Table

In [131]: tab
Out[131]: /ints_floats (Table(0,)) 'Integers and Floats'
             description := {
             "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
             "No1": Int32Col(shape=(), dflt=0, pos=1),
             "No2": Int32Col(shape=(), dflt=0, pos=2),
             "No3": Float64Col(shape=(), dflt=0.0, pos=3),
             "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
             byteorder := 'little'
             chunkshape := (2621,)
```

❶ 以 HDF5 二进制存储格式打开数据库文件。

❷ `Date` 列用于日期-时间信息（字符串对象形式）。

❸ 保存整数对象的两列。

❹ 保存浮点对象的两列。

❺ 通过 `Filters` 对象可以指定压缩级别。

❻ 表的节点（路径）和技术名称。

❼ 行数据结构描述。

❽ 表名（标题）。

❾ 预期的行数；可以优化。

❿ 用于该表的 `Filters` 对象。

为了在表中填入数值数据，我们生成了两个包含随机数的 `ndarray` 对象：一个包含随机整数，另一个包含随机浮点数。填充表的工作由一个简单的 Python 循环完成：

```
In [132]: pointer = tab.row ❶

In [133]: ran_int = np.random.randint(0, 10000, size=(rows, 2)) ❷

In [134]: ran_flo = np.random.standard_normal((rows, 2)).round(4) ❸

In [135]: %%time
          for i in range(rows):
              pointer['Date'] = dt.datetime.now() ❹
              pointer['No1'] = ran_int[i, 0] ❹
              pointer['No2'] = ran_int[i, 1] ❹
              pointer['No3'] = ran_flo[i, 0] ❹
              pointer['No4'] = ran_flo[i, 1] ❹
              pointer.append() ❺
          tab.flush() ❻
          CPU times: user 8.16 s, sys: 78.7 ms, total: 8.24 s
          Wall time: 8.25 s

In [136]: tab ❼
Out[136]: /ints_floats (Table(2000000,)) 'Integers and Floats'
            description := {
            "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
            "No1": Int32Col(shape=(), dflt=0, pos=1),
            "No2": Int32Col(shape=(), dflt=0, pos=2),
            "No3": Float64Col(shape=(), dflt=0.0, pos=3),
            "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
            byteorder := 'little'
            chunkshape := (2621,)

In [137]: ll $path*
          -rw-r--r-- 1 yves staff 100156248 Oct 19 12:12
          /Users/yves/Temp/data/pytab.h5
```

❶ 创建一个指针对象。

❷ 创建包含随机整数的 `ndarray` 对象。

❸ 创建包含随机浮点数的 `ndarray` 对象。

❹ 逐行写入 `datetime` 对象、两个整数和两个浮点数对象。

❺ 附加新行。

❻ 所有写入行都必须“冲走”（flushed），也就是提交为永久性更改。

❼ 更改反映在 `Table` 对象的描述中。

本例中的 Python 循环相当慢。通过使用 NumPy 结构数组，我们可以以性能更好、更具 Python

风格的方式实现相同的结果。将完整数据集保存在结构数组中后，创建表就只需要一行代码。注意，不再需要行描述。PyTables 使用结构数组的 dtype 对象来推导数据类型：

```
In [138]: dty = np.dtype([('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'),
                                          ('No3', '<f8'), ('No4', '<f8')]) ❶

In [139]: sarray = np.zeros(len(ran_int), dtype=dty) ❷

In [140]: sarray[:4] ❸
Out [140]: array([(b'', 0, 0, 0., 0.), (b'', 0, 0, 0., 0.), (b'', 0, 0, 0., 0.),
                  (b'', 0, 0, 0., 0.)],
          dtype=[('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'), ('No3', '<f8'),
          ('No4', '<f8')])

In [141]: %%time
          sarray['Date'] = dt.datetime.now() ❹
          sarray['No1'] = ran_int[:, 0] ❹
          sarray['No2'] = ran_int[:, 1] ❹
          sarray['No3'] = ran_flo[:, 0] ❹
          sarray['No4'] = ran_flo[:, 1] ❹
          CPU times: user 161 ms, sys: 42.7 ms, total: 204 ms
          Wall time: 207 ms

In [142]: %%time
          h5.create_table('/', 'ints_floats_from_array', sarray,
                                title='Integers and Floats',
                                expectedrows=rows, filters=filters) ❺
          CPU times: user 42.9 ms, sys: 51.4 ms, total: 94.3 ms
          Wall time: 96.6 ms

Out[142]: /ints_floats_from_array (Table(2000000,)) 'Integers and Floats'
           description := {
           "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
           "No1": Int32Col(shape=(), dflt=0, pos=1),
           "No2": Int32Col(shape=(), dflt=0, pos=2),
           "No3": Float64Col(shape=(), dflt=0.0, pos=3),
           "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
           byteorder := 'little'
           chunkshape := (2621,)
```

❶ 定义特殊 dtype 对象。

❷ 创建包含 0（和空串）的结构数组。

❸ 来自 ndarray 对象的几个记录。

❹ 一次性填充 ndarray 对象各列。

❺ 创建 `Table` 对象并填入数据。

这种方法的速度会提升一个数量级，从而以更简洁的代码实现相同结果：

```
In [143]: type(h5)
Out[143]: tables.file.File

In [144]: h5 ❶
Out[144]: File(filename=/Users/yves/Temp/data/pytab.h5, title='', mode='w',
          root_uep='/', filters=Filters(complevel=0, shuffle=False,
          bitshuffle=False, fletcher32=False, least_significant_digit=None))
         / (RootGroup) ''
         /ints_floats (Table(2000000,)) 'Integers and Floats'
           description := {
           "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
           "No1": Int32Col(shape=(), dflt=0, pos=1),
           "No2": Int32Col(shape=(), dflt=0, pos=2),
           "No3": Float64Col(shape=(), dflt=0.0, pos=3),
           "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
           byteorder := 'little'
           chunkshape := (2621,)
         /ints_floats_from_array (Table(2000000,)) 'Integers and Floats'
           description := {
           "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
           "No1": Int32Col(shape=(), dflt=0, pos=1),
           "No2": Int32Col(shape=(), dflt=0, pos=2),
           "No3": Float64Col(shape=(), dflt=0.0, pos=3),
           "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
           byteorder := 'little'
           chunkshape := (2621,)

In [145]: h5.remove_node('/', 'ints_floats_from_array') ❷
```

❶ 包含两个 `Table` 对象的 `File` 对象描述。

❷ 删除包含冗余数据的第二个 `Table` 对象。

`Table` 对象在大部分情况下与 NumPy 结构化数组的表现很相似（见图 9-5）：

```
In [146]: tab[:3] ❶
Out [146]: array([(b'2018-10-19 12:12:28.227771', 8576, 5991, -0.0528, 0.2468),
                  (b'2018-10-19 12:12:28.227858', 2990, 9310, -0.0261, 0.3932),
                  (b'2018-10-19 12:12:28.227868', 4400, 4823, 0.9133, 0.2579)],
          dtype=[('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'), ('No3', '<f8'),
         ('No4', '<f8')])

In [147]: tab[:4]['No4'] ❷
```

```
Out[147]: array([ 0.2468, 0.3932, 0.2579, -0.5582])

In [148]: %time np.sum(tab[:]['No3']) ❸
          CPU times: user 76.7 ms, sys: 74.8 ms, total: 151 ms
          Wall time: 152 ms

Out[148]: 88.8542999999997

In [149]: %time np.sum(np.sqrt(tab[:]['No1'])) ❸
          CPU times: user 91 ms, sys: 57.9 ms, total: 149 ms
          Wall time: 164 ms

Out[149]: 133349920.3689251

In [150]: %%time
          plt.figure(figsize=(10, 6))
          plt.hist(tab[:]['No3'], bins=30); ❹
          CPU times: user 328 ms, sys: 72.1 ms, total: 400 ms
          Wall time: 456 ms
```

❶ 通过索引选择行。

❷ 通过索引选择列值。

❸ 应用 NumPy 通用函数。

❹ 根据 `Table` 对象的一列绘制图表。

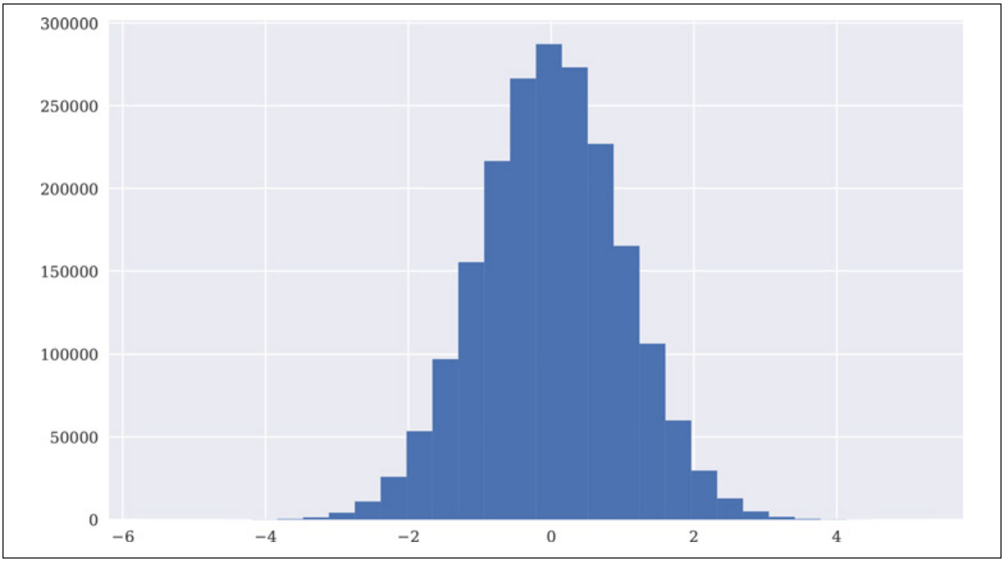


图 9-5　列数据的直方图

PyTables 还提供了通过典型类 SQL 语句查询数据的灵活工具，如下面的例子（结果如图 9-6 所示；可与基于 pandas 查询的图 9-2 进行对比）：

```
In [151]: query = '((No3 < -0.5) | (No3 > 0.5)) & ((No4 < -1) | (No4 > 1))'❶

In [152]: iterator = tab.where(query)❷

In [153]: %time res = [(row['No3'], row['No4']) for row in iterator]❸
          CPU times: user 269 ms, sys: 64.4 ms, total: 333 ms
          Wall time: 294 ms

In [154]: res = np.array(res) ❹
          res[:3]
Out[154]: array([[0.7694, 1.4866],
                 [0.9201, 1.3346],
                 [1.4701, 1.8776]])
In [155]: plt.figure(figsize=(10, 6))
          plt.plot(res.T[0], res.T[1], 'ro');
```

❶ 查询采用字符串对象的形式，4 个条件用逻辑运算符组合。

❷ 基于查询的迭代器对象。

❸ 从查询得到的数据行通过列表推导被收集。

❹ 转换为 ndarray 对象。

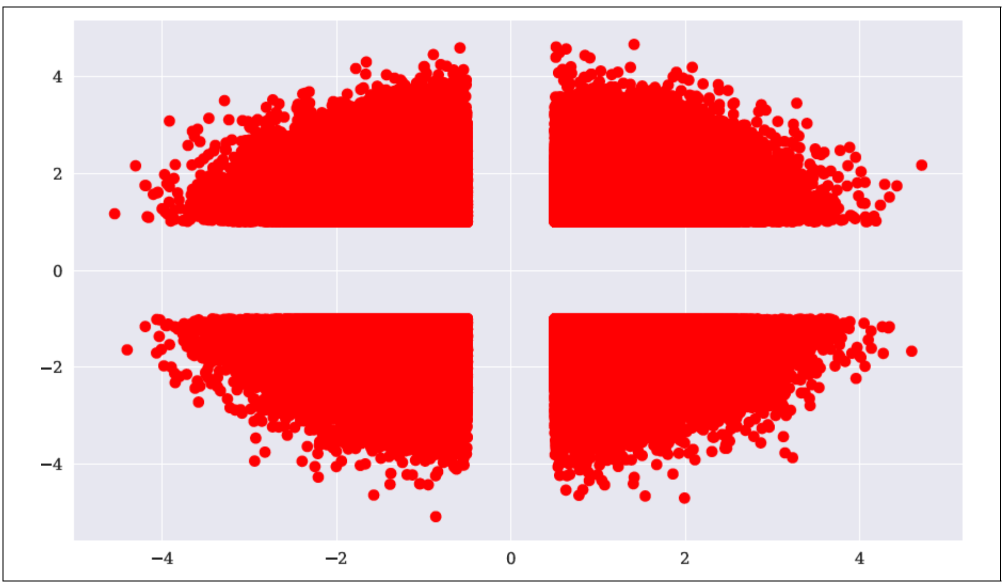


图 9-6　列数据的直方图

快速查询

pandas 和 PyTables 都可以处理相对复杂的类 SQL 查询和选择。它们在这些操作上都对速度进行了优化。虽然这些方法与关系数据库相比还有一些局限性，但对于大部分数值和金融应用都无伤大雅。

正如下面的例子所示，从语法和性能的角度看，以表（Table）对象的形式使用 PyTables 中保存的数据，都给人一种在内存中使用 NumPy 或者 pandas 对象的印象：

```
In [156]: %%time
          values = tab[:]['No3']
          print('Max %18.3f' % values.max())
          print('Ave %18.3f' % values.mean())
          print('Min %18.3f' % values.min())
          print('Std %18.3f' % values.std())
          Max              5.224
          Ave              0.000
          Min             -5.649
          Std              1.000
          CPU times: user 163 ms, sys: 70.4 ms, total: 233 ms
          Wall time: 234 ms

In [157]: %%time
          res = [(row['No1'], row['No2']) for row in
                  tab.where('((No1 > 9800) | (No1 < 200)) \
                          & ((No2 > 4500) & (No2 < 5500))')]
          CPU times: user 165 ms, sys: 52.5 ms, total: 218 ms
          Wall time: 155 ms

In [158]: for r in res[:4]:
              print(r)
          (91, 4870)
          (9803, 5026)
          (9846, 4859)
          (9823, 5069)

In [159]: %%time
          res = [(row['No1'], row['No2']) for row in
                  tab.where('(No1 == 1234) & (No2 > 9776)')]
          CPU times: user 58.9 ms, sys: 40.5 ms, total: 99.4 ms
          Wall time: 81 ms

In [160]: for r in res:
              print(r)
```

```
(1234, 9841)
(1234, 9821)
(1234, 9867)
(1234, 9987)
(1234, 9849)
(1234, 9800)
```

9.3.2 使用压缩表

使用 PyTables 的主要优势之一是压缩方法。使用压缩不仅能节约磁盘空间，还能改善 I/O 操作的性能。这是如何实现的呢？当 I/O 成为瓶颈而 CPU 能够快速（解）压缩数据时，使用压缩对速度有正面的净效应。下面的例子基于最新型（在编写本书时）SSD 的 I/O 操作，因此没有观察到压缩的速度优势。但是，使用压缩几乎没有任何劣势：

```
In [161]: filename = path + 'pytabc.h5'

In [162]: h5c = tb.open_file(filename, 'w')

In [163]: filters = tb.Filters(complevel=5, ❶
                               complib='blosc') ❷

In [164]: tabc = h5c.create_table('/', 'ints_floats', sarray,
                                  title='Integers and Floats',
                                  expectedrows=rows, filters=filters)

In [165]: query = '((No3 < -0.5) | (No3 > 0.5)) & ((No4 < -1) | (No4 > 1))'

In [166]: iteratorc = tabc.where(query) ❸

In [167]: %time res = [(row['No3'], row['No4']) for row in iteratorc]❹
          CPU times: user 300 ms, sys: 50.8 ms, total: 351 ms
          Wall time: 311 ms

In [168]: res = np.array(res)
          res[:3]
Out[168]: array([[0.7694, 1.4866],
                 [0.9201, 1.3346],
                 [1.4701, 1.8776]])
```

❶ `complevel`（压缩级别）参数取值范围为 `0`（无压缩）到 `9`（最高压缩率）。

❷ 使用 Blosc 压缩引擎，该引擎优化了性能。

❸ 根据之前的查询来创建迭代器对象。

❹ 查询得到的数据行可通过列表推导进行收集。

生成包含原始数据的压缩表并在上面进行分析，其速度比在未压缩的表上稍慢。那么读入 ndarray 会如何呢？我们检查一下：

```
In [169]: %time arr_non = tab.read() ❶
          CPU times: user 63 ms, sys: 78.5 ms, total: 142 ms
          Wall time: 149 ms

In [170]: tab.size_on_disk
Out[170]: 100122200

In [171]: arr_non.nbytes
Out[171]: 100000000

In [172]: %time arr_com = tabc.read() ❷
          CPU times: user 106 ms, sys: 55.5 ms, total: 161 ms
          Wall time: 173 ms

In [173]: tabc.size_on_disk
Out[173]: 41306140

In [174]: arr_com.nbytes
Out[174]: 100000000

In [175]: ll $path* ❸
          -rw-r--r-- 1 yves staff 200312336 Oct 19 12:12
          /Users/yves/Temp/data/pytab.h5
          -rw-r--r-- 1 yves staff 41341436 Oct 19 12:12
          /Users/yves/Temp/data/pytabc.h5

In [176]: h5c.close() ❹
```

❶ 从未压缩的表对象 tab 读取。

❷ 从压缩表对象 tabc 读取。

❸ 比较大小——压缩表的尺寸明显减小。

❹ 关闭数据库文件。

上述例子说明，与未压缩的表相比，使用压缩表对象在速度上几乎没有任何差别。但是磁盘文件尺寸可能显著减小（取决于数据质量），这有几个优点：

- 降低存储成本；
- 降低备份成本；
- 降低网络流量；
- 提高网络速度（远程服务器存储及检索更快）；

- 增大 CPU 利用率以克服 I/O 瓶颈。

9.3.3 使用数组

在 9.1 节中我们已经看到，NumPy 内建 ndarray 对象的快速写入和读取能力。PyTables 在存储和检索 ndarray 对象时也相当快速、高效。由于它基于层次数据库结构，所以具备许多方便的特性：

```
In [177]: %%time
          arr_int = h5.create_array('/', 'integers', ran_int) ❶
          arr_flo = h5.create_array('/', 'floats', ran_flo) ❷
          CPU times: user 4.26 ms, sys: 37.2 ms, total: 41.5 ms
          Wall time: 46.2 ms

In [178]: h5❸
Out[178]: File(filename=/Users/yves/Temp/data/pytab.h5, title='', mode='w',
          root_uep='/', filters=Filters(complevel=0, shuffle=False,
          bitshuffle=False, fletcher32=False, least_significant_digit=None))
         / (RootGroup) ''
         /floats (Array(2000000, 2)) ''
           atom := Float64Atom(shape=(), dflt=0.0)
           maindim := 0
           flavor := 'numpy'
           byteorder := 'little'
           chunkshape := None
         /integers (Array(2000000, 2)) ''
           atom := Int64Atom(shape=(), dflt=0)
           maindim := 0
           flavor := 'numpy'
           byteorder := 'little'
           chunkshape := None
         /ints_floats (Table(2000000,)) 'Integers and Floats'
           description := {
           "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
           "No1": Int32Col(shape=(), dflt=0, pos=1),
           "No2": Int32Col(shape=(), dflt=0, pos=2),
           "No3": Float64Col(shape=(), dflt=0.0, pos=3),
           "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
           byteorder := 'little'
           chunkshape := (2621,)

In [179]: ll $path*
          -rw-r--r-- 1 yves staff 262344490 Oct 19 12:12
           /Users/yves/Temp/data/pytab.h5
          -rw-r--r-- 1 yves staff 41341436 Oct 19 12:12
```

```
          /Users/yves/Temp/data/pytabc.h5

In [180]: h5.close()

In [181]: !rm -f $path*
```

❶ 保存 `ndarray` 对象 `ran_int`

❷ 保存 `ndarray` 对象 `ran_flo`。

❸ 在对象描述中反映更改。

将这些对象直接写入 HDF5 数据库，比循环读取对象、然后逐行写入表对象或者使用结构化 `ndarray` 对象的方法都要快。

> **基于 HDF5 的数据存储**
>
> HDF5 数据库（文件）格式是结构化数值和金融数据的强大替代方案，可以用它来代替关系数据库。在使用 PyTables 时单独访问或者结合 pandas 的能力，都可以得到硬件所能支持的最高 I/O 性能。

9.3.4 内存外计算

PyTables 支持内存外计算，因此可以实现不适合于内存的基于数组的计算。为此，考虑以下基于 EArray 类的代码。这种对象可在一维（行方向）上扩展，但列数（每行元素数）必须固定：

```
In [182]: filename = path + 'earray.h5'

In [183]: h5 = tb.open_file(filename, 'w')

In [184]: n = 500 ❶

In [185]: ear = h5.create_earray('/', 'ear', ❷
                                 atom=tb.Float64Atom(), ❸
                                 shape=(0, n)) ❹

In [186]: type(ear)
Out[186]: tables.earray.Earray

In [187]: rand = np.random.standard_normal((n, n)) ❺
          rand[:4, :4]
Out[187]: array([[-1.25983231, 1.11420699, 0.1667485 , 0.7345676 ],
                 [-0.13785424, 1.22232417, 1.36303097, 0.13521042],
                 [ 1.45487119, -1.47784078, 0.15027672, 0.86755989],
```

```
                    [-0.63519366, 0.1516327 , -0.64939447, -0.45010975]])

In [188]: %%time
          for _ in range(750):
              ear.append(rand) ❻
          ear.flush()
          CPU times: user 814 ms, sys: 1.18 s, total: 1.99 s
          Wall time: 2.53 s

In [189]: ear
Out[189]: /ear (EArray(375000, 500)) ''
            atom := Float64Atom(shape=(), dflt=0.0)
            maindim := 0
            flavor := 'numpy'
            byteorder := 'little'
            chunkshape := (16, 500)

In [190]: ear.size_on_disk
Out[190]: 1500032000
```

❶ 固定的列数。

❷ `EArray` 对象的路径和技术名称。

❸ 单个值的原子 `dtype` 对象。

❹ 实例组成（`0` 行，`n` 列）。

❺ 包含随机数的 `ndarray` 对象。

❻ 进行多次附加操作。

对于没有导致聚合的内存外计算，需要另一个同样组成（大小）的 `EArray` 对象。PyTables 有一个特殊模块，可以高效处理数值表达式。这个模块叫作 Expr，它基于数值表达式库 `numexpr`。下面的代码使用 `Expr`，在上例中的整个 `EArray` 对象上计算公式 9-1 所示的数学表达式。

公式 9-1　数学表达式示例

$$y = 3\sin(x) + \sqrt{|x|}$$

结果保存在 `EArray` 对象 `out` 中，表达式的求值成批进行：

```
In [191]: out = h5.create_earray('/', 'out',
                                 atom=tb.Float64Atom(),
                                 shape=(0, n))

In [192]: out.size_on_disk
```

```
Out[192]: 0

In [193]: expr = tb.Expr('3 * sin(ear) + sqrt(abs(ear))')❶

In [194]: expr.set_output(out, append_mode=True)❷

In [195]: %time expr.eval()❸
          CPU times: user 3.08 s, sys: 1.7 s, total: 4.78 s
          Wall time: 4.03 s

Out[195]: /out (EArray(375000, 500)) ''
            atom := Float64Atom(shape=(), dflt=0.0)
            maindim := 0
            flavor := 'numpy'
            byteorder := 'little'
            chunkshape := (16, 500)

In [196]: out.size_on_disk
Out[196]: 1500032000

In [197]: out[0, :10]
Out[197]: array([-1.73369462, 3.74824436, 0.90627898, 2.86786818,
           1.75424957,
          -0.91108973, -1.68313885, 1.29073295, -1.68665599, -1.71345309])

In [198]: %time out_ = out.read() ❹
          CPU times: user 1.03 s, sys: 1.1 s, total: 2.13 s
          Wall time: 2.22 s

In [199]: out_[0, :10]
Out[199]: array([-1.73369462, 3.74824436, 0.90627898, 2.86786818,
           1.75424957,
          -0.91108973, -1.68313885, 1.29073295, -1.68665599, -1.71345309])
```

❶ 将一个基于表达式的 `str` 对象转换为 `Expr` 对象。

❷ 定义输出为 `EArray` 对象 `out`。

❸ 初始化表达式的值。

❹ 将整个 `EArray` 读入内存。

考虑到整个运算在内存外进行，这样的结果应该算相当快速了，尤其是在标准硬件上执行时。我们简单地将其与 `numexpr` 模块在内存中的性能做个比较（参见第 10 章）。

速度提高了，但不是非常巨大的提高：

```
In [200]: import numexpr as ne ❶

In [201]: expr = '3 * sin(out_) + sqrt(abs(out_))'❷

In [202]: ne.set_num_threads(1)❸
Out[202]: 4

In [203]: %time ne.evaluate(expr)[0, :10] ❹
          CPU times: user 2.51 s, sys: 1.54 s, total: 4.05 s
          Wall time: 4.94 s

Out[203]: array([-1.64358578, 0.22567882, 3.31363043, 2.50443549,
           4.27413965,
          -1.41600606, -1.68373023, 4.01921805, -1.68117412, -1.66053597])

In [204]: ne.set_num_threads(4) ❺
Out[204]: 1
In [205]: %time ne.evaluate(expr)[0, :10]❻
          CPU times: user 3.39 s, sys: 1.94 s, total: 5.32 s
          Wall time: 2.96 s

Out[205]: array([-1.64358578, 0.22567882, 3.31363043, 2.50443549,
           4.27413965,
          -1.41600606, -1.68373023, 4.01921805, -1.68117412, -1.66053597])

In [206]: h5.close()

In [207]: !rm -f $path*
```

❶ 导入在内存中求数值表达式的值所用的模块。

❷ 字符串对象形式的数值表达式。

❸ 将线程数设置为 1。

❹ 通过单个线程在内存中求数值表达式的值。

❺ 将线程数设置为 4。

❻ 通过 4 个线程在内存中求数值表达式的值。

9.4 TsTables 的 I/O

TsTables 软件包使用 PyTables 来为时间序列数据构建高性能存储。主要使用场景是“一次写入，多次检索”。这在金融分析中是典型的场景：数据在市场中创建，实时或者异步读

取，并存储在磁盘上供以后使用。该数据可能用于较大的交易策略验证程序，这种程序需要一次又一次地使用历史金融时间序列数据的不同子集，此时快速的数据检索非常重要。

9.4.1 样板数据

和往常一样，第一个任务是生成足够大的样板数据集，以说明 TsTables 的优势。以下的代码基于对几何布朗运动（参见第 12 章）的模拟生成了 3 个相当长的金融时间序列：

```
In [208]: no = 5000000 ❶
          co = 3 ❷
          interval = 1. / (12 * 30 * 24 * 60) ❸
          vol = 0.2 ❹
In [209]: %%time
          rn = np.random.standard_normal((no, co)) ❺
          rn[0] = 0.0 ❻
          paths = 100 * np.exp(np.cumsum(-0.5 * vol ** 2 * interval +
                  vol * np.sqrt(interval) * rn, axis=0)) ❼
          paths[0] = 100 ❽
          CPU times: user 869 ms, sys: 175 ms, total: 1.04 s
          Wall time: 812 ms
```

❶ 时间步数。

❷ 时间序列数量。

❸ 以年表示的时间间隔。

❹ 波动率。

❺ 标准正态分布随机数。

❻ 将初始随机数设置为 0。

❼ 基于欧拉离散的模拟。

❽ 将初始路径值设置为 100。

由于 `TsTables` 能够很好地处理 pandas 的 `DataFrame` 对象，所以我们将数据转换成这样的对象（见图 9-7）：

```
In [210]: dr = pd.date_range('2019-1-1', periods=no, freq='1s')

In [211]: dr[-6:]
Out[211]: DatetimeIndex(['2019-02-27 20:53:14', '2019-02-27 20:53:15',
                         '2019-02-27 20:53:16', '2019-02-27 20:53:17',
                         '2019-02-27 20:53:18', '2019-02-27 20:53:19'],
                        dtype='datetime64[ns]', freq='S')

In [212]: df = pd.DataFrame(paths, index=dr, columns=['ts1', 'ts2', 'ts3'])
```

```
In [213]: df.info()
          <class 'pandas.core.frame.DataFrame'>
          DatetimeIndex: 5000000 entries, 2019-01-01 00:00:00 to 2019-02-27
           20:53:19
          Freq: S
          Data columns (total 3 columns):
          ts1     float64
          ts2     float64
          ts3     float64
          dtypes: float64(3)
          memory usage: 152.6 MB
In [214]: df.head()
Out[214]:                            ts1         ts2         ts3
          2019-01-01 00:00:00 100.000000 100.000000 100.000000
          2019-01-01 00:00:01 100.018443  99.966644  99.998255
          2019-01-01 00:00:02 100.069023 100.004420  99.986646
          2019-01-01 00:00:03 100.086757 100.000246  99.992042
          2019-01-01 00:00:04 100.105448 100.036033  99.950618

In [215]: df[::100000].plot(figsize=(10, 6));
```



图 9-7　在金融时间序列中选择的数据点

9.4.2　数据存储

TsTables 的金融时间序列数据的存储基于特殊的块式结构，能够快速检索由某种时间

间隔定义的任意数据子集。为此，该软件包为 PyTables 增加了函数 `create_ts()`。下面我们使用基于 PyTables 中 `tb.IsDescription` 类的方法，提供表各列的数据类型：

```
In [216]: import tstables as tstab

In [217]: class ts_desc(tb.IsDescription):
              timestamp = tb.Int64Col(pos=0) ❶
              ts1 = tb.Float64Col(pos=1) ❷
              ts2 = tb.Float64Col(pos=2) ❷
              ts3 = tb.Float64Col(pos=3) ❷
In [218]: h5 = tb.open_file(path + 'tstab.h5', 'w') ❸

In [219]: ts = h5.create_ts('/', 'ts', ts_desc) ❹

In [220]: %time ts.append(df) ❺
          CPU times: user 1.36 s, sys: 497 ms, total: 1.86 s
          Wall time: 1.29 s

In [221]: type(ts)
Out[221]: tstables.tstable.TsTable

In [222]: ls -n $path
          total 328472
          -rw-r--r-- 1 501 20 157037368 Oct 19 12:13 tstab.h5
```

❶ 时间戳数据列。

❷ 保存数值数据的列。

❸ 打开 HDF5 数据库文件以便写入（`w`）。

❹ 根据 `ts_desc` 对象创建 `TsTable` 对象。

❺ 将 `DataFrame` 对象的数据附加到 `TsTable` 对象。

9.4.3 数据检索

尽管依赖于硬件，但用 TsTables 写入数据的速度明显很快。将大块数据读回内存时也一样。很方便的是，`TsTables` 返回一个 `DataFrame` 对象（见图 9-8）：

```
In [223]: read_start_dt = dt.datetime(2019, 2, 1, 0, 0) ❶
          read_end_dt = dt.datetime(2019, 2, 5, 23, 59) ❷

In [224]: %time rows = ts.read_range(read_start_dt, read_end_dt) ❸
          CPU times: user 182 ms, sys: 73.5 ms, total: 255 ms
          Wall time: 163 ms

In [225]: rows.info() ❹
          <class 'pandas.core.frame.DataFrame'>
```

```
DatetimeIndex: 431941 entries, 2019-02-01 00:00:00 to 2019-02-05
  23:59:00
Data columns (total 3 columns):
ts1    431941 non-null float64
ts2    431941 non-null float64
ts3    431941 non-null float64
dtypes: float64(3)
memory usage: 13.2 MB

In [226]: rows.head() ❹
Out[226]:                          ts1        ts2         ts3
          2019-02-01 00:00:00 52.063640 40.474580 217.324713
          2019-02-01 00:00:01 52.087455 40.471911 217.250070
          2019-02-01 00:00:02 52.084808 40.458013 217.228712
          2019-02-01 00:00:03 52.073536 40.451408 217.302912
          2019-02-01 00:00:04 52.056133 40.450951 217.207481

In [227]: h5.close()

In [228]: (rows[::500] / rows.iloc[0]).plot(figsize=(10, 6));
```

❶ 区间起始时间。

❷ 区间结束时间。

❸ `ts.read_range()`函数返回区间的 `DataFrame` 对象。

❹ `DataFrame` 对象有几十万行数据。



图 9-8　特定时间段的金融时间序列（已规格化）

为了更好地说明基于 TsTables 数据检索的性能，考虑如下基准程序，该程序读取 100 块数据，这些数据由 4 天的秒数据组成。读取包含 345 600 行数据的 DataFrame 对象花费的时间不到 0.1s：

```
In [229]: import random

In [230]: h5 = tb.open_file(path + 'tstab.h5', 'r')

In [231]: ts = h5.root.ts._f_get_timeseries()❶

In [232]: %%time
          for _ in range(100): ❷
              d = random.randint(1, 24) ❸
              read_start_dt = dt.datetime(2019, 2, d, 0, 0, 0)
              read_end_dt = dt.datetime(2019, 2, d + 3, 23, 59, 59)
              rows = ts.read_range(read_start_dt, read_end_dt)
          CPU times: user 7.17 s, sys: 1.65 s, total: 8.81 s
          Wall time: 4.78 s

In [233]: rows.info() ❹
          <class 'pandas.core.frame.DataFrame'>
          DatetimeIndex: 345600 entries, 2019-02-04 00:00:00 to 2019-02-07
            23:59:59
          Data columns (total 3 columns):
          ts1    345600 non-null float64
          ts2    345600 non-null float64
          ts3    345600 non-null float64
          dtypes: float64(3)
          memory usage: 10.5 MB

In [234]: !rm $path/tstab.h5
```

❶ 连接到 TsTable 对象。

❷ 数据检索重复多次。

❸ 随机化起始日值。

❹ 读取最后一个 DataFrame 对象。

9.5 结语

基于 SQL（即关系）的数据库在单个对象/表格之间存在大量关系的复杂数据结构上具

有优势，这可以为某些情况下它们的性能劣于基于 ndarray 的纯 NumPy 方法或者基于 DataFrame 的 pandas 方法辩解。

不过，金融或者科学中的许多应用领域通常使用基于数组的数据建模方法。在这些情况下，使用原生 NumPy I/O 功能、结合 NumPy 和 PyTables 的能力，或者通过基于 HDF5 存储的 pandas 方法，可以实现巨大的性能改进。

`TsTable` 在处理大型（金融）时间序列数据集时特别有用，尤其在“一次写入，多次检索”的场景中。

虽然近来的趋势是使用基于云的解决方案——云由大量基于商用硬件的计算节点组成——但是人们应该小心考虑，特别是在金融领域，哪一种硬件架构最适合于分析需求。Microsoft 最近的一项研究阐明了这一主题：

> 我们断言，单个“垂直扩容”服务器能够处理所有此类工作，并且在性能、成本、电源消耗和服务器密度上与群集相当或者更好。
>
> ——Appuswamy 等人（2013）

因此，参与数据分析的公司、研究机构和其他组织应该首先分析需要完成的特定任务，然后决定根据以下方式决定硬件/软件架构。

水平扩容

　　使用具有许多商用节点的群集，每个节点采用标准 CPU 和相对较少的内存。

垂直扩容

　　使用一个或者少数强大的具有多核 CPU 的服务器，可能它还有一个 GPU 以及大量内存，甚至在需要机器学习和深度学习起作用时还有 TPU。

垂直扩容硬件并应用不同的实现方法可能会显著影响性能，下一章将更详细地介绍这个主题。

9.6 延伸阅读

本章开始和结束中引用的论文是很好的读物，也是思考金融分析硬件架构的很好起点。

Appuswamy, Raja, et al. (2013). *Nobody Ever Got Fired for Buying a Cluster*. Microsoft Technical Report.

和往常一样，Web 提供了与本章介绍的主题及 Python 软件包相关的许多有价值资源：

- 用 pickle 进行的 Python 对象序列化参见 Python 官方文档；
- SciPy 网站提供了 NumPy I/O 能力的概述；

- pandas 在线文档的 I/O 章节；
- PyTables 首页提供的教程和详细文档；
- TsTables 的更多信息可以参见其 GitHub 页面。

在 GitHub yhilpisch 中可以找到 TsTables 的一个友好的分支。安装这个分支中的软件包，这些软件包维护新版本的 pandas 和其他 Python 软件包的兼容性。

第 10 章

高性能的 Python

不要降低预期去屈就性能，而要提升性能满足预期。

——拉尔夫 · 马斯顿

有一种长期存在的偏见，认为 Python 本身是一种相对低效的编程语言，不适合用来处理金融业务的计算任务。除了 Python 是一种解释型语言之外，出现这种偏见的推理方式通常是：Python 处理循环时很慢，金融算法的实现往往需要循环，因此，Python 对于金融算法的实现来说太慢了。另一种推理方式是：其他（编译型）语言（如 C 或者 C++）可以快速执行循环，金融算法常常需要循环，因此，这些（编译型）编程语言适合于金融业和金融算法。

必须承认，即便写出合适的 Python 代码，也有可能执行得很慢——可能对于许多应用领域太慢。本章介绍的是加速金融环境下常见典型任务及算法的方法。本章将介绍，如何明智地使用数据结构、选择合适的实现方法及范型，以及使用正确的高性能软件包。Python 甚至可以与编译型编程语言匹敌，这是因为它本身也是编译后执行的。

为此，本章将介绍让代码加速的不同方法。

向量化

　　利用 Python 的向量化功能，是前几章已经广泛使用的方法。

动态编译

　　Numba 软件包可以让我们使用 LLVM 技术来动态地编译纯 Python 代码。

静态编译

　　Cython 不仅是一个 Python 软件包，还是一种组合了 Python 和 C 的混合语言。例如，它可以使用静态类型声明，静态编译经过调整的代码。

多进程

Python 的 `multiprocessing` 模块可以简便地实现代码并行执行。

本章介绍如下内容。

循环

本节介绍 Python 循环及其加速方法。

算法

本节讲解的是常用作性能基准的标准数学算法，例如斐波那契数列的生成。

二叉树

二项式期权定价模型是广泛使用的金融模型，可作为更复杂金融算法的一个案例分析。

蒙特卡洛模拟

类似地，蒙特卡洛模拟在金融实践中广泛用于定价和风险管理。这种算法对计算的要求很高，长期被视为 C 或者 C++语言的“领地”。

pandas 递归算法

本节介绍基于金融时间序列数据的递归算法的加速。特别地，它还介绍了计算指数加权移动平均数（EWMA）的不同实现方法。

10.1　循环

本节解决 Python 的循环问题。任务相当简单：编写一个函数提取“大量”随机数，然后返回平均值。我们感兴趣的是执行时间，这可以用神奇的`%time` 和`%timeit` 函数估算。

10.1.1　Python

让我们“慢慢”地开始——原谅我的双关语。在纯 Python 中，这样的函数可能像下面的 `average_py()`：

```
In [1]: import random

In [2]: def average_py(n):
            s = 0 ❶
            for i in range(n):
                s += random.random() ❷
            return s / n ❸

In [3]: n = 10000000 ❹
```

```
In [4]: %time average_py(n) ❺
        CPU times: user 1.82 s, sys: 10.4 ms, total: 1.83 s
        Wall time: 1.93 s

Out[4]: 0.5000590124747943

In [5]: %timeit average_py(n) ❻
        1.31 s ± 159 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [6]: %time sum([random.random() for _ in range(n)]) / n ❼
        CPU times: user 1.55 s, sys: 188 ms, total: 1.74 s
        Wall time: 1.74 s

Out[6]: 0.49987031710661173
```

❶ 初始化变量 s 的值。

❷ 在 s 中添加区间（0,1）的均匀分布随机值。

❸ 返回平均值。

❹ 定义循环次数。

❺ 测定函数执行一次的时间。

❻ 多次测定函数执行时间，得到更可靠的估算。

❼ 使用列表推导代替函数。

这为以后的其他方法设置了基准。

10.1.2　NumPy

NumPy 的优势在于其向量化能力。从形式上看，Python 级别的循环消失了；循环在更深的一级上，由 NumPy 提供的优化和预编译例程执行[1]。average_np()函数利用了这种方法：

```
In [7]: import numpy as np

In [8]: def average_np(n):
            s = np.random.random(n) ❶
            return s.mean() ❷

In [9]: %time average_np(n)
        CPU times: user 180 ms, sys: 43.2 ms, total: 223 ms
        Wall time: 224 ms
```

1　NumPy 也可以使用专用的数学库，如 Intel 数学核心库。——原注

```
Out[9]: 0.49988861556468317

In [10]: %timeit average_np(n)
         128 ms ± 2.01 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [11]: s = np.random.random(n)
         s.nbytes ❸
Out[11]: 80000000
```

❶ “一次性”提取随机数（没有 Python 循环）。

❷ 返回平均值。

❸ 创建的 `ndarray` 对象使用的字节数。

加速很可观，几乎达到 10 倍（一个数量级）。然而，为此必须付出的代价是明显更高的内存使用率。这是因为 NumPy 依靠预先分配数据并在编译后的层次内处理来得到高速度。因此，用这种方法无法处理“流化”数据。不同的算法或者问题使用的内存量甚至可能更高。

> **向量化与内存**
> 在一切可能的情况下编写向量化代码是很有诱惑力的，因为它具备简洁的语法，速度通常也有所加快。不过，这些好处也是有代价的，通常带来更大的内存占用。

10.1.3 Numba

Numba 软件包通过使用 LLVM 可以动态地编译纯 Python 代码。在简单的情况下（如以下的代码），它的应用非常直观，用于动态编译的 `average_nb()`函数可以从 Python 直接调用：

```
In [12]: import numba

In [13]: average_nb = numba.jit(average_py) ❶

In [14]: %time average_nb(n) ❷
         CPU times: user 204 ms, sys: 34.3 ms, total: 239 ms
         Wall time: 278 ms

Out[14]: 0.4998865391283664

In [15]: %time average_nb(n) ❸
         CPU times: user 80.9 ms, sys: 457 µs, total: 81.3 ms
         Wall time: 81.7 ms
```

```
Out[15]: 0.5001357454250273

In [16]: %timeit average_nb(n)
         75.5 ms ± 1.95 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

❶ 创建 Numba 函数。

❷ 编译发生在运行时，会带来一些开销。

❸ 从第二次执行（使用相同的输入数据类型）起，执行更快了。

纯 Python 和 Numba 的组合战胜了 NumPy 版本，并且保留了基于循环的原始实现的内存效率。还有一点很明显，在这种简单情况下应用 Numba 几乎没有任何编程开销。

> **没有免费的午餐**
>
> 对比 Python 代码和编译版本的性能，Numba 的应用有时就像魔法，特别是它还很易于使用。不过，许多情况下 Numba 并不适用，性能增进几乎难以察觉，甚至无法实现。

10.1.4 Cython

Cython 可以静态编译 Python 代码。但是，它的应用不像 Numba 那么简单，通常需要更改代码才能看到明显的加速。首先，考虑 Cython 函数 `average_cy1()`，该函数为使用的变量引入了静态类型声明：

```
In [17]: %load_ext Cython

In [18]: %%cython -a
         import random ❶
         def average_cy1(int n): ❷
             cdef int i ❷
             cdef float s = 0 ❷
             for i in range(n):
                 s += random.random()
             return s / n
Out[18]: <IPython.core.display.HTML object>

In [19]: %time average_cy1(n)
         CPU times: user 695 ms, sys: 4.31 ms, total: 699 ms
         Wall time: 711 ms

Out[19]: 0.49997106194496155

In [20]: %timeit average_cy1(n)
         752 ms ± 91.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

❶ 在 Cython 上下文里导入 `random` 模块。

❷ 为变量 `n`、`i` 和 `s` 添加静态类型声明。

可以观察到一些加速，但还没有达到之前介绍的方法（如 NumPy 版本）取得的效果。有必要对 Cython 略作优化，这样它甚至可以击败 Numba 版本：

```
In [21]: %%cython
         from libc.stdlib cimport rand ❶
         cdef extern from 'limits.h': ❷
             int INT_MAX ❷
         cdef int i
         cdef float rn
         for i in range(5):
             rn = rand() / INT_MAX ❸
             print(rn)
         0.6792964339256287
         0.934692919254303
         0.3835020661354065
         0.5194163918495178
         0.8309653401374817
In [22]: %%cython -a
         from libc.stdlib cimport rand ❶
         cdef extern from 'limits.h': ❷
             int INT_MAX ❷
         def average_cy2(int n):
             cdef int i
             cdef float s = 0
             for i in range(n):
                 s += rand() / INT_MAX ❸
             return s / n
Out[22]: <IPython.core.display.HTML object>

In [23]: %time average_cy2(n)
         CPU times: user 78.5 ms, sys: 422 µs, total: 79 ms
         Wall time: 79.1 ms

Out[23]: 0.500017523765564

In [24]: %timeit average_cy2(n)
         65.4 ms ± 706 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

❶ 从 C 中导入一个随机数生成器。

❷ 导入一个恒定值，对随机数按比例进行调整。

❸ 比例调整后，添加区间（0,1）中的均匀分布随机数。

进一步优化的 Cython 版本函数 `average_cy2()`此时略快于 Numba 版本。但是，所花的精力也比较多。与 NumPy 版本相比，Cython 还保留了基于循环实现的内存效率。

> **Cython = Python + C**
>
> Cython 使开发人员可以尽可能地调整代码以提高性能，也可以明智地少做调整——例如，从纯 Python 版本入手，向代码中加入越来越多的 C 语言元素。在编译步骤中也可以设置参数，以进一步优化编译后的版本。

10.2 算法

本节将上一节的性能增强技术应用到一些著名的数学问题和算法上。这些算法经常用作性能基准。

10.2.1 质数

质数不仅在理论数学中起到了重要作用，在许多计算机科学应用学科（如加密）中也是如此。质数是大于 1 且只能被 1 和自身整除的自然数，除此之外没有其他因子。较大的质数很稀有，寻找起来很难，但很容易证明某个数不是质数。唯一需要的条件就是找到大于 1 的自然数因子。

1. Python

测试数值是不是质数有许多算法可以实现。下面是算法一个 Python 版本，虽然它从算法角度说不是最优的，但也相当高效。不过，对于较大的质数 `p2`，其执行时间很长：

```
In [25]: def is_prime(I):
             if I % 2 == 0: return False ❶
             for i in range(3, int(I ** 0.5) + 1, 2): ❷
                 if I % i == 0: return False ❸
             return True ❹

In [26]: n = int(1e8 + 3) ❺
         n
Out[26]: 100000003

In [27]: %time is_prime(n)
         CPU times: user 35 µs, sys: 0 ns, total: 35 µs
         Wall time: 39.1 µs

Out[27]: False
```

```
In [28]: p1 = int(1e8 + 7) ❺
         p1
Out[28]: 100000007

In [29]: %time is_prime(p1)
         CPU times: user 776 µs, sys: 1 µs, total: 777 µs
         Wall time: 787 µs

Out[29]: True

In [30]: p2 = 100109100129162907 ❻

In [31]: p2.bit_length() ❻
Out[31]: 57

In [32]: %time is_prime(p2)
         CPU times: user 22.6 s, sys: 44.7 ms, total: 22.6 s
         Wall time: 22.7 s
Out[32]: True
```

❶ 如果是偶数，立即返回 `False`。

❷ 循环从 3 开始，直到 `I` 的平方根+1，步长为 2。

❸ 确定一个因子之后，函数返回 `False`。

❹ 如果没有找到因子，返回 `True`。

❺ 相对小的非质数和质数。

❻ 较大的质数需要更长的执行时间。

2. Numba

函数 `is_prime()`中的算法循环结构由 Numba 动态编译。开销仍然很少，而加速效果很可观：

```
In [33]: is_prime_nb = numba.jit(is_prime)

In [34]: %time is_prime_nb(n) ❶
         CPU times: user 87.5 ms, sys: 7.91 ms, total: 95.4 ms
         Wall time: 93.7 ms

Out[34]: False

In [35]: %time is_prime_nb(n) ❷
```

```
        CPU times: user 9 µs, sys: 1e+03 ns, total: 10 µs
        Wall time: 13.6 µs

Out[35]: False

In [36]: %time is_prime_nb(p1)
         CPU times: user 26 µs, sys: 0 ns, total: 26 µs
         Wall time: 31 µs

Out[36]: True

In [37]: %time is_prime_nb(p2) ❸
         CPU times: user 1.72 s, sys: 9.7 ms, total: 1.73 s
         Wall time: 1.74 s

Out[37]: True
```

❶ 第一次调用 `is_prime_nb()`涉及编译开销。

❷ 从第二次调用起，可以较为全面地看到加速的效果。

❸ 较大质数的速度提高了大约一个数量级。

3. Cython

Cython 的应用也很简单。没有类型声明的普通 Cython 版本已经明显加快了执行代码的速度：

```
In [38]: %%cython
         def is_prime_cy1(I):
             if I % 2 == 0: return False
             for i in range(3, int(I ** 0.5) + 1, 2):
                 if I % i == 0: return False
             return True

In [39]: %timeit is_prime(p1)
         394 µs ± 14.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [40]: %timeit is_prime_cy1(p1)
         243 µs ± 6.58 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

不过，真正的改进只有使用静态类型声明才能实现。此时，Cython 版本甚至稍快于 Numba：

```
In [41]: %%cython
         def is_prime_cy2(long I): ❶
             cdef long i ❶
             if I % 2 == 0: return False
             for i in range(3, int(I ** 0.5) + 1, 2):
```

```
                if I % i == 0: return False
            return True

In [42]: %timeit is_prime_cy2(p1)
         87.6 µs ± 27.7 µs per loop (mean ± std. dev. of 7 runs,
         10000 loops each)

In [43]: %time is_prime_nb(p2)
         CPU times: user 1.68 s, sys: 9.73 ms, total: 1.69 s
         Wall time: 1.7 s

Out[43]: True

In [44]: %time is_prime_cy2(p2)
         CPU times: user 1.66 s, sys: 9.47 ms, total: 1.67 s
         Wall time: 1.68 s

Out[44]: True
```

❶ 变量 `I` 和 `i` 的静态类型声明。

4. 多进程

到目前为止，所有的优化工作都集中在代码的顺序执行上。特别是对于质数，可能需要同时检查多个数值。在这方面，`multiprocessing` 模块有助于进一步加速代码执行。你可以用它产生多个并行运行的 Python 进程。对于简单问题，该模块的应用很直观。首先，将 `mp.Pool` 对象设置为多进程。其次，将要执行的函数映射到所要检查的质数上：

```
In [45]: import multiprocessing as mp

In [46]: pool = mp.Pool(processes=4) ❶

In [47]: %time pool.map(is_prime, 10 * [p1]) ❷
         CPU times: user 1.52 ms, sys: 2.09 ms, total: 3.61 ms
         Wall time: 9.73 ms

Out[47]: [True, True, True, True, True, True, True, True, True, True]

In [48]: %time pool.map(is_prime_nb, 10 * [p2]) ❷
         CPU times: user 13.9 ms, sys: 4.8 ms, total: 18.7 ms
         Wall time: 10.4 s

Out[48]: [True, True, True, True, True, True, True, True, True, True]

In [49]: %time pool.map(is_prime_cy2, 10 * [p2]) ❷
```

```
        CPU times: user 9.8 ms, sys: 3.22 ms, total: 13 ms
        Wall time: 9.51 s

Out[49]: [True, True, True, True, True, True, True, True, True, True]
```

❶ 将 `mp.Pool` 对象实例化为多进程。

❷ 对应的函数映射到包含质数的列表对象上。

加速效果很显著，Python 函数 `is_prime()`对于较大的质数 `p2` 需要花费 20 多秒的时间。在 4 个进程并行执行时，`is_prime_nb()`和 `is_prime_cy2()`函数都可以在不到 10 秒的时间内完成 10 倍于 `p2` 的质数。

> **并行处理**
>
> 每当需要解决多个相同类型问题时，就应该考虑并行处理。如果拥有具备多个核心及充足工作内存的强大硬件，效果可能非常巨大。`multiprocessing` 是标准库中的一个易用模块。

10.2.2　斐波那契数

斐波那契数（以及数列）可以由一个简单算法得出。从两个 1 开始，第 3 个数（下一个斐波那契数）是前两个数的和：1，1，2，3，5，8，13，21，…。本节会分析该数列的两种不同实现：递归算法和循环（迭代）算法。

1. 递归算法

众所周知，与常规的 Python 循环类似，Python 的常规递归函数实现也相对较慢。这样的函数可能需要多次调用自身，以得出最后的结果。下面的 `fib_rec_py1()`函数就是这样的实现。在本例中，Numba 完全无法加速执行。不过，Cython 仅依靠静态类型声明就显著加快了执行速度：

```
In [50]: def fib_rec_py1(n):
             if n < 2:
                 return n
             else:
                 return fib_rec_py1(n - 1) + fib_rec_py1(n - 2)

In [51]: %time fib_rec_py1(35)
         CPU times: user 6.55 s, sys: 29 ms, total: 6.58 s
         Wall time: 6.6 s

Out[51]: 9227465

In [52]: fib_rec_nb = numba.jit(fib_rec_py1)
```

```
In [53]: %time fib_rec_nb(35)
         CPU times: user 3.87 s, sys: 24.2 ms, total: 3.9 s
         Wall time: 3.91 s

Out[53]: 9227465

In [54]: %%cython
         def fib_rec_cy(int n):
             if n < 2:
                 return n
             else:
                 return fib_rec_cy(n - 1) + fib_rec_cy(n - 2)

In [55]: %time fib_rec_cy(35)
         CPU times: user 751 ms, sys: 4.37 ms, total: 756 ms
         Wall time: 755 ms

Out[55]: 9227465
```

递归算法的主要问题是中间结果不会缓存，而是重新计算。为了避免出现这种特有的问题，可以使用一个装饰器（decorator）来负责缓存中间结果。它可以将执行速度提高好几个数量级：

```
In [56]: from functools import lru_cache as cache

In [57]: @cache(maxsize=None) ❶
         def fib_rec_py2(n):
             if n < 2:
                 return n
             else:
                 return fib_rec_py2(n - 1) + fib_rec_py2(n - 2)

In [58]: %time fib_rec_py2(35) ❷
         CPU times: user 64 µs, sys: 28 µs, total: 92 µs
         Wall time: 98 µs

Out[58]: 9227465

In [59]: %time fib_rec_py2(80) ❷
         CPU times: user 38 µs, sys: 8 µs, total: 46 µs
         Wall time: 51 µs

Out[59]: 23416728348467685
```

❶ 缓存中间结果。

❷ 该操作带来极大的加速效果。

2. 循环（迭代）算法

虽然计算第 n 个斐波那契数的算法可以用递归方法实现，但并不是必须这么做。接下来介绍一个循环（迭代）实现，即便以纯 Python 编写，也比递归实现的缓存版本快。这也是 Numba 可以进一步改善性能的领域。不过，Cython 版本是最终的胜者：

```
In [60]: def fib_it_py(n):
             x, y = 0, 1
             for i in range(1, n + 1):
                 x, y = y, x + y
             return x

In [61]: %time fib_it_py(80)
         CPU times: user 19 µs, sys: 1e+03 ns, total: 20 µs
         Wall time: 26 µs

Out[61]: 23416728348467685
In [62]: fib_it_nb = numba.jit(fib_it_py)

In [63]: %time fib_it_nb(80)
         CPU times: user 57 ms, sys: 6.9 ms, total: 63.9 ms
         Wall time: 62 ms

Out[63]: 23416728348467685

In [64]: %time fib_it_nb(80)
         CPU times: user 7 µs, sys: 1 µs, total: 8 µs
         Wall time: 12.2 µs

Out[64]: 23416728348467685

In [65]: %%cython
         def fib_it_cy1(int n):
             cdef long i
             cdef long x = 0, y = 1
             for i in range(1, n + 1):
                 x, y = y, x + y
             return x

In [66]: %time fib_it_cy1(80)
         CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
         Wall time: 11 µs

Out[66]: 23416728348467685
```

既然一切都这么快了，你可能会感到奇怪，我们为什么只计算第 80 个斐波那契数，而不是更后面的（例如，第 150 个）。问题出在可用的数据类型上。虽然 Python 本质上可以处理任意大的数值（参见 3.1 节），但编译语言通常不能。不过，Cython 可以依靠特殊数据类型，得到比 64 位双精度浮点数对象所允许的更大的数值：

```
In [67]: %%time
         fn = fib_rec_py2(150) ❶
         print(fn) ❶
         9969216677189303386214405760200
         CPU times: user 361 µs, sys: 115 µs, total: 476 µs
         Wall time: 430 µs

In [68]: fn.bit_length() ❷
Out[68]: 103

In [69]: %%time
         fn = fib_it_nb(150) ❸
         print(fn) ❸
         6792540214324356296
         CPU times: user 270 µs, sys: 78 µs, total: 348 µs
         Wall time: 297 µs
In [70]: fn.bit_length() ❹
Out[70]: 63

In [71]: %%time
         fn = fib_it_cy1(150) ❸
         print(fn) ❸
         6792540214324356296
         CPU times: user 255 µs, sys: 71 µs, total: 326 µs
         Wall time: 279 µs

In [72]: fn.bit_length() ❹
Out[72]: 63

In [73]: %%cython
         cdef extern from *:
             ctypedef int int128 '__int128_t' ❺
         def fib_it_cy2(int n):
             cdef int128 i ❺
             cdef int128 x = 0, y = 1 ❺
             for i in range(1, n + 1):
                 x, y = y, x + y
             return x

In [74]: %%time
```

```
         fn = fib_it_cy2(150) ❻
         print(fn) ❻
         9969216677189303386214405760200
         CPU times: user 280 µs, sys: 115 µs, total: 395 µs
         Wall time: 328 µs

In [75]: fn.bit_length() ❻
Out[75]: 103
```

❶ Python 版本快速而正确。

❷ 结果整数的位数为 103（大于 64）。

❸ Numba 和 Cython 版本比较快，但结果不正确。

❹ 由于 64 位整数对象的限制，遇到了溢出问题。

❺ 导入特殊的 128 位整数对象类型并使用。

❻ `fib_it_cy2()`的 Cython 版本现在更快且正确。

10.2.3 π

本节分析的最后一个算法是基于蒙特卡洛模拟的 π 值求取算法。[1]基本思路依据这样的事实：圆面积 A 由公式 $A=\pi r^2$ 给出，因此 $\pi=\frac{A}{r^2}$。对于半径 r=1 的单位元，π=A。这个算法的思路是模拟随机点的坐标值（x,y），$x, y \in [-1, 1]$。以圆点为中心、边长为 2 的正方形的面积为 4。以圆点为中心的单位圆是这个正方形区域的一部分，这部分的比例可以通过蒙特卡洛模拟估算：为正方形内的所有点计数，然后为圆内的点计数，最后用圆内点计数除以正方形内点计数。下面的例子展示了这一算法（参见图 10-1）：

```
In [76]: import random
         import numpy as np
         from pylab import mpl, plt
         plt.style.use('seaborn')
         mpl.rcParams['font.family'] = 'serif'
         %matplotlib inline

In [77]: rn = [(random.random() * 2 - 1, random.random() * 2 - 1)
               for _ in range(500)]

In [78]: rn = np.array(rn)
         rn[:5]
Out[78]: array([[ 0.45583018, -0.27676067],
```

1 例子的灵感来自 Code Review Stack Exchange 网站上的帖子。——原注

```
                [-0.70120038, 0.15196888],
                [ 0.07224045, 0.90147321],
                [-0.17450337, -0.47660912],
                [ 0.94896746, -0.31511879]])

In [79]: fig = plt.figure(figsize=(7, 7))
         ax = fig.add_subplot(1, 1, 1)
         circ = plt.Circle((0, 0), radius=1, edgecolor='g', lw=2.0,
                           facecolor='None') ❶
         box = plt.Rectangle((-1, -1), 2, 2, edgecolor='b', alpha=0.3) ❷
         ax.add_patch(circ) ❶
         ax.add_patch(box) ❷
         plt.plot(rn[:, 0], rn[:, 1], 'r.') ❸
         plt.ylim(-1.1, 1.1)
         plt.xlim(-1.1, 1.1)
```

❶ 绘制单位圆。

❷ 绘制边长为 2 的正方形。

❸ 绘制均匀分布的随机点。

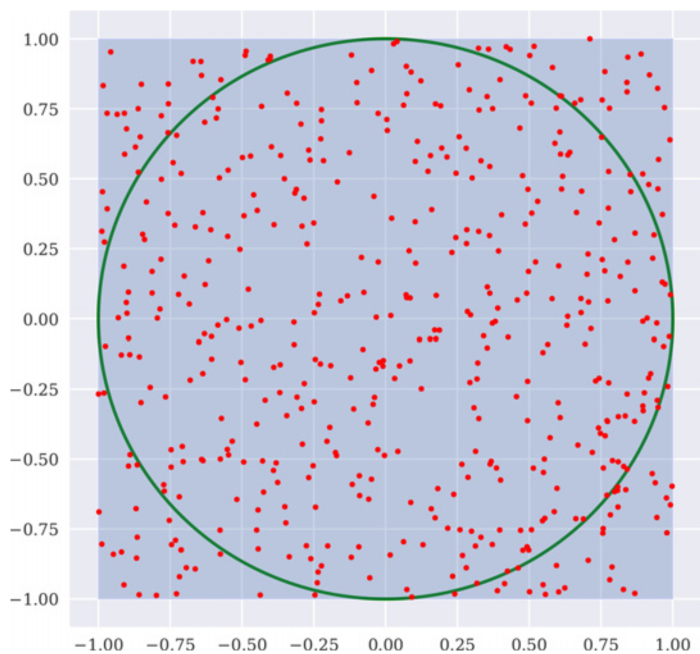


图 10-1　包含均匀分布随机点的单位圆及边长为 2 的正方形

这一算法的 NumPy 实现相当简洁，但也是内存密集型。给定参数的总执行时间大约

为 1 秒：

```
In [80]: n = int(1e7)

In [81]: %time rn = np.random.random((n, 2)) * 2 - 1
         CPU times: user 450 ms, sys: 87.9 ms, total: 538 ms
         Wall time: 573 ms

In [82]: rn.nbytes
Out[82]: 160000000

In [83]: %time distance = np.sqrt((rn ** 2).sum(axis=1)) ❶
         distance[:8].round(3)
         CPU times: user 537 ms, sys: 198 ms, total: 736 ms
         Wall time: 651 ms

Out[83]: array([1.181, 1.061, 0.669, 1.206, 0.799, 0.579, 0.694, 0.941])

In [84]: %time frac = (distance <= 1.0).sum() / len(distance) ❷
         CPU times: user 47.9 ms, sys: 6.77 ms, total: 54.7 ms
         Wall time: 28 ms

In [85]: pi_mcs = frac * 4 ❸
         pi_mcs ❸
Out[85]: 3.1413396
```

❶ 点与原点的距离（欧几里得范数）。

❷ 圆内点数与所有点数的比例。

❸ 这个比例乘以正方形面积 4 就是圆面积的估算值，即 π 值。

Python 函数 `mcs_pi_py()`通过一个 `for` 循环，以高效利用内存的方式实现了蒙特卡洛模拟。注意，随机数在这里没有做比例调整。执行时间比 NumPy 版本长，本例中 Numba 版本的执行速度快于 NumPy：

```
In [86]: def mcs_pi_py(n):
             circle = 0
             for _ in range(n):
                 x, y = random.random(), random.random()
                 if (x ** 2 + y ** 2) ** 0.5 <= 1:
                     circle += 1
             return (4 * circle) / n

In [87]: %time mcs_pi_py(n)
         CPU times: user 5.47 s, sys: 23 ms, total: 5.49 s
         Wall time: 5.43 s
```

```
Out[87]: 3.1418964

In [88]: mcs_pi_nb = numba.jit(mcs_pi_py)

In [89]: %time mcs_pi_nb(n)
         CPU times: user 319 ms, sys: 6.36 ms, total: 326 ms
         Wall time: 326 ms

Out[89]: 3.1422012

In [90]: %time mcs_pi_nb(n)
         CPU times: user 284 ms, sys: 3.92 ms, total: 288 ms
         Wall time: 291 ms

Out[90]: 3.142066
```

仅使用静态类型声明的简单 Cython 版本比 Python 版本快不了多少。但是，我们可以再次依靠 C 的随机数生成能力来进一步加速计算：

```
In [91]: %%cython -a
         import random
         def mcs_pi_cy1(int n):
             cdef int i, circle = 0
             cdef float x, y
             for i in range(n):
                 x, y = random.random(), random.random()
                 if (x ** 2 + y ** 2) ** 0.5 <= 1:
                     circle += 1
             return (4 * circle) / n
Out[91]: <IPython.core.display.HTML object>

In [92]: %time mcs_pi_cy1(n)
         CPU times: user 1.15 s, sys: 8.24 ms, total: 1.16 s
         Wall time: 1.16 s

Out[92]: 3.1417132

In [93]: %%cython -a
         from libc.stdlib cimport rand
         cdef extern from 'limits.h':
             int INT_MAX
         def mcs_pi_cy2(int n):
             cdef int i, circle = 0
             cdef float x, y
```

```
            for i in range(n):
                x, y = rand() / INT_MAX, rand() / INT_MAX
                if (x ** 2 + y ** 2) ** 0.5 <= 1:
                    circle += 1
            return (4 * circle) / n
Out[93]: <IPython.core.display.HTML object>

In [94]: %time mcs_pi_cy2(n)
         CPU times: user 170 ms, sys: 1.45 ms, total: 172 ms
         Wall time: 172 ms

Out[94]: 3.1419388
```

算法类型

本节分析的算法可能与金融算法没有直接关联。不过，这些算法的优点在于简单、易于理解。此外，金融环境下的典型算法问题可以在这种简化的环境下讨论。

10.3　二叉树

二项式期权定价模型是一种流行的期权定价数值方法，是由 Cox、Ross 和 Rubinstein 率先提出的。这种方法以一棵（重组）树来表示某种资产可能的未来发展。这种模型的设置和 Black-Scholes-Merton 的设置一样，有风险资产（指数或者股票）以及无风险资产（债券）。当天与期权到期日之间的时间通常被分为等距的子间隔 Δt。如果时间 s 的指数水平为 S_s，则 $t = s+\Delta t$ 时的指数水平为 $S_t = S_s \cdot m$，其中 m 从{u，d}中随机选取（$0 < d < e^{r\Delta t} < u = e^{\sigma\sqrt{\Delta t}}$，$u$=1/$d$）。$r$ 是一个常数——无风险短期利率。

10.3.1　Python

下面的代码是一个 Python 实现，即根据固定的模型数值参数来创建一棵重组树：

```
In [95]: import math

In [96]: S0 = 36. ❶
         T = 1.0 ❷
         r = 0.06 ❸
         sigma = 0.2 ❹

In [97]: def simulate_tree(M):
             dt = T / M ❺
             u = math.exp(sigma * math.sqrt(dt)) ❻
             d = 1 / u ❻
```

```
    S = np.zeros((M + 1, M + 1))
    S[0, 0] = S0
    z = 1
    for t in range(1, M + 1):
        for i in range(z):
            S[i, t] = S[i, t-1] * u
            S[i+1, t] = S[i, t-1] * d
        z += 1
    return S
```

❶ 风险资产初始价值。

❷ 二叉树模拟的时间范围。

❸ 恒定的短期利率。

❹ 恒定的波动率因数。

❺ 时间间隔。

❻ 向上和向下运动因数。

与典型的树图不同，向上运动在 `ndarray` 对象中表现为侧向运动，这显著减小了 `ndarray` 的大小：

```
In [98]: np.set_printoptions(formatter={'float':
                                         lambda x: '%6.2f' % x})

In [99]: simulate_tree(4) ❶
Out[99]: array([[ 36.00, 39.79, 43.97, 48.59, 53.71],
                [ 0.00, 32.57, 36.00, 39.79, 43.97],
                [ 0.00, 0.00, 29.47, 32.57, 36.00],
                [ 0.00, 0.00, 0.00, 26.67, 29.47],
                [ 0.00, 0.00, 0.00, 0.00, 24.13]])

In [100]: %time simulate_tree(500) ❷
          CPU times: user 148 ms, sys: 4.49 ms, total: 152 ms
          Wall time: 154 ms

Out[100]: array([[ 36.00, 36.32, 36.65, ..., 3095.69, 3123.50, 3151.57],
                 [ 0.00, 35.68, 36.00, ..., 3040.81, 3068.13, 3095.69],
                 [ 0.00, 0.00, 35.36, ..., 2986.89, 3013.73, 3040.81],
                 ...,
                 [ 0.00, 0.00, 0.00, ..., 0.42, 0.42, 0.43],
                 [ 0.00, 0.00, 0.00, ..., 0.00, 0.41, 0.42],
                 [ 0.00, 0.00, 0.00, ..., 0.00, 0.00, 0.41]])
```

❶ 4 个时间间隔的树。

❷ 500 个时间间隔的树。

10.3.2 NumPy

利用一些技巧，上述二叉树可用 NumPy 全向量化代码创建：

```
In [101]: M = 4

In [102]: up = np.arange(M + 1)
          up = np.resize(up, (M + 1, M + 1)) ❶
          up
Out[102]: array([[0, 1, 2, 3, 4],
                 [0, 1, 2, 3, 4],
                 [0, 1, 2, 3, 4],
                 [0, 1, 2, 3, 4],
                 [0, 1, 2, 3, 4]])

In [103]: down = up.T * 2 ❷
          down
Out[103]: array([[0, 0, 0, 0, 0],
                 [2, 2, 2, 2, 2],
                 [4, 4, 4, 4, 4],
                 [6, 6, 6, 6, 6],
                 [8, 8, 8, 8, 8]])

In [104]: up – down ❸
Out[104]: array([[ 0, 1, 2, 3, 4],
                 [-2, -1, 0, 1, 2],
                 [-4, -3, -2, -1, 0],
                 [-6, -5, -4, -3, -2],
                 [-8, -7, -6, -5, -4]])

In [105]: dt = T / M

In [106]: S0 * np.exp(sigma * math.sqrt(dt) * (up - down)) ❹
Out[106]: array([[ 36.00, 39.79, 43.97, 48.59, 53.71],
                 [ 29.47, 32.57, 36.00, 39.79, 43.97],
                 [ 24.13, 26.67, 29.47, 32.57, 36.00],
                 [ 19.76, 21.84, 24.13, 26.67, 29.47],
                 [ 16.18, 17.88, 19.76, 21.84, 24.13]])
```

❶ 包含总向上运动的 `ndarray` 对象。

❷ 包含总向下运动的 `ndarray` 对象。

❸ 包含净向上（正）和向下运动的 `ndarray` 对象。

❹ 4 个时间间隔的树（值的右上方三角形）。

在 NumPy 中，代码略微紧凑一些。但更重要的是，NumPy 向量化实现了一个数量级的加速且没有使用更多内存：

```
In [107]: def simulate_tree_np(M):
              dt = T / M
              up = np.arange(M + 1)
              up = np.resize(up, (M + 1, M + 1))
              down = up.transpose() * 2
              S = S0 * np.exp(sigma * math.sqrt(dt) * (up - down))
              return S

In [108]: simulate_tree_np(4)
Out[108]: array([[ 36.00, 39.79, 43.97, 48.59, 53.71],
                 [ 29.47, 32.57, 36.00, 39.79, 43.97],
                 [ 24.13, 26.67, 29.47, 32.57, 36.00],
                 [ 19.76, 21.84, 24.13, 26.67, 29.47],
                 [ 16.18, 17.88, 19.76, 21.84, 24.13]])

In [109]: %time simulate_tree_np(500)
          CPU times: user 8.72 ms, sys: 7.07 ms, total: 15.8 ms
          Wall time: 12.9 ms

Out[109]: array([[ 36.00, 36.32, 36.65, ..., 3095.69, 3123.50, 3151.57],
                 [ 35.36, 35.68, 36.00, ..., 3040.81, 3068.13, 3095.69],
                 [ 34.73, 35.05, 35.36, ..., 2986.89, 3013.73, 3040.81],
                 ...,
                 [ 0.00, 0.00, 0.00, ..., 0.42, 0.42, 0.43],
                 [ 0.00, 0.00, 0.00, ..., 0.41, 0.41, 0.42],
                 [ 0.00, 0.00, 0.00, ..., 0.40, 0.41, 0.41]])
```

10.3.3 Numba

这种金融算法应该很适合于通过 Numba 动态编译来优化。确实，可以观察到比 NumPy 版本有一个数量级的加速。这样，Numba 版本比 Python（或者混合）版本要快上好几个数量级：

```
In [110]: simulate_tree_nb = numba.jit(simulate_tree)

In [111]: simulate_tree_nb(4)
Out[111]: array([[ 36.00, 39.79, 43.97, 48.59, 53.71],
                 [  0.00, 32.57, 36.00, 39.79, 43.97],
                 [  0.00,  0.00, 29.47, 32.57, 36.00],
                 [  0.00,  0.00,  0.00, 26.67, 29.47],
                 [  0.00,  0.00,  0.00,  0.00, 24.13]])

In [112]: %time simulate_tree_nb(500)
```

```
CPU times: user 425 µs, sys: 193 µs, total: 618 µs
Wall time: 625 µs

Out[112]: array([[ 36.00, 36.32, 36.65, ..., 3095.69, 3123.50, 3151.57],
                [  0.00, 35.68, 36.00, ..., 3040.81, 3068.13, 3095.69],
                [  0.00,  0.00, 35.36, ..., 2986.89, 3013.73, 3040.81],
                ...,
                [  0.00,  0.00,  0.00, ...,    0.42,    0.42,    0.43],
                [  0.00,  0.00,  0.00, ...,    0.00,    0.41,    0.42],
                [  0.00,  0.00,  0.00, ...,    0.00,    0.00,    0.41]])

In [113]: %timeit simulate_tree_nb(500)
          559 µs ± 46.1 µs per loop (mean ± std. dev. of 7 runs, 1000
          loops each)
```

10.3.4 Cython

和前面一样，Cython 需要对代码进行更多的调整，才能看到显著的改进。以下版本主要使用静态类型声明，并导入了一些额外的功能模块。与常规的 Python 导入及函数相比，这些模块的性能有所改善：

```
In [114]: %%cython -a
          import numpy as np
          cimport cython
          from libc.math cimport exp, sqrt
          cdef float S0 = 36.
          cdef float T = 1.0
          cdef float r = 0.06
          cdef float sigma = 0.2
          def simulate_tree_cy(int M):
              cdef int z, t, i
              cdef float dt, u, d
              cdef float[:, :] S = np.zeros((M + 1, M + 1),
                                             dtype=np.float32) ❶
              dt = T / M
              u = exp(sigma * sqrt(dt))
              d = 1 / u
              S[0, 0] = S0
              z = 1
              for t in range(1, M + 1):
                  for i in range(z):
                      S[i, t] = S[i, t-1] * u
                      S[i+1, t] = S[i, t-1] * d
                  z += 1
              return np.array(S)
Out[114]: <IPython.core.display.HTML object>
```

❶ 将 `ndarray` 对象声明为 C 数组，这对性能至关重要。

与 Numba 版本相比，Cython 版本将执行时间又缩短了 30%：

```
In [115]: simulate_tree_cy(4)
Out[115]: array([[ 36.00, 39.79, 43.97, 48.59, 53.71],
                 [  0.00, 32.57, 36.00, 39.79, 43.97],
                 [  0.00,  0.00, 29.47, 32.57, 36.00],
                 [  0.00,  0.00,  0.00, 26.67, 29.47],
                 [  0.00,  0.00,  0.00,  0.00, 24.13]], dtype=float32)

In [116]: %time simulate_tree_cy(500)
          CPU times: user 2.21 ms, sys: 1.89 ms, total: 4.1 ms
          Wall time: 2.45 ms

Out[116]: array([[ 36.00, 36.32, 36.65, ..., 3095.77, 3123.59, 3151.65],
                 [ 0.00, 35.68, 36.00, ..., 3040.89, 3068.21, 3095.77],
                 [ 0.00, 0.00, 35.36, ..., 2986.97, 3013.81, 3040.89],
                 ...,
                 [ 0.00, 0.00, 0.00, ..., 0.42, 0.42, 0.43],
                 [ 0.00, 0.00, 0.00, ..., 0.00, 0.41, 0.42],
                 [ 0.00, 0.00, 0.00, ..., 0.00, 0.00, 0.41]],
                 dtype=float32)
In [117]: %timeit S = simulate_tree_cy(500)
          363 µs ± 29.5 µs per loop (mean ± std. dev. of 7 runs, 1000
          loops each)
```

10.4　蒙特卡洛模拟

蒙特卡洛模拟是计算金融学中不可或缺的数值工具，在现代计算机出现之前就已经投入使用。银行和其他金融机构将其用于定价和风险管理，这种方法可能是金融学中最为灵活和强大的数值方法之一。不过，它也往往是计算要求最高的。因此，Python 长期被排除在实现基于蒙特卡洛模拟算法的合适编程语言之外——至少在现实应用场景里是这样。

本节分析几何布朗运动的蒙特卡洛模拟，这是一种简单但广泛应用的随机过程，用于建立股价或者指数水平演变的模型。Black-Scholes-Merton 期权定价理论（1973）借鉴了这一过程。在这种设置下，所要估值的期权标的遵循随机微分方程式（SDE），如公式 10-1 所示。S_t 是时间 t 的标的价值；r 是一个常数——无风险短期利率；σ 是恒定瞬时波动率；Z_t 是布朗运动。

公式 10-1　Black-Scholes-Merton SDE（几何布朗运动）

$$dS_t = rS_t dt + \sigma S_t dZ_t$$

这个 SDE 可以在等距时间间隔上离散化，并根据公式 10-2 模拟，公式 10-2 以欧拉格式表示。其中，z 是标准正态分布随机数。对于 M 个时间间隔，其长度表示为 $\Delta t = \frac{T}{M}$，其中 T 为模拟的时间范围（例如，所要估值的期权到期日）。

公式 10-2　Black-Scholes-Merton 差分方程（欧拉格式）

$$S_t = S_{t-\Delta t} \exp\left(\left(r - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t} z\right)$$

欧式看涨期权的蒙特卡洛估算函数在公式 10-3 中给出，其中 $S_T(i)$是到期日 T 时第 i 个标的的模拟价值，模拟路径总数为 I（i=1,2，⋯，I）。

公式 10-3　欧式看涨期权的蒙特卡洛估算函数

$$C_0 = \mathrm{e}^{-rT} \frac{1}{I} \sum_I \max(S_T(i) - K, 0)$$

10.4.1　Python

首先是按照公式 10-2 实现蒙特卡洛模拟的 Python（或者混合）版本——`mcs_simulation_py()`。之所以说它是个混合版本，是因为在 `ndarray` 对象上执行 Python 循环。前面已经看到，这可能为 Numba 动态编译打下很好的基础。和往常一样，将这个函数的执行时间作为基准。根据模拟来估算欧洲看跌期权的价值：

```
In [118]: M = 100 ❶
          I = 50000 ❷

In [119]: def mcs_simulation_py(p):
              M, I = p
              dt = T / M
              S = np.zeros((M + 1, I))
              S[0] = S0
              rn = np.random.standard_normal(S.shape) ❸
              for t in range(1, M + 1): ❹
                  for i in range(I): ❹
                      S[t, i] = S[t-1, i] * math.exp((r - sigma ** 2 /
                      2) * dt + sigma * math.sqrt(dt) * rn[t, i]) ❹
              return S

In [120]: %time S = mcs_simulation_py((M, I))
          CPU times: user 5.55 s, sys: 52.9 ms, total: 5.6 s
          Wall time: 5.62 s

In [121]: S[-1].mean() ❺
```

```
Out[121]: 38.22291254503985

In [122]: S0 * math.exp(r * T)❻
Out[122]: 38.22611567563295

In [123]: K = 40. ❼

In [124]: C0 = math.exp(-r * T) * np.maximum(K - S[-1], 0).mean() ❽

In [125]: C0 # ❽
Out[125]: 3.860545188088036
```

❶ 离散化的时间间隔数量。

❷ 模拟的路径数量。

❸ 用单一向量化步骤提取的随机数。

❹ 实现基于欧拉格式模拟的嵌套循环。

❺ 根据模拟算出的期末价值均值。

❻ 理论预期期末价值。

❼ 欧洲看跌期权行权价。

❽ 蒙特卡洛期权估值函数。

图 10-2 展示了模拟期末（欧洲看跌期权到期日）模拟价值的直方图。

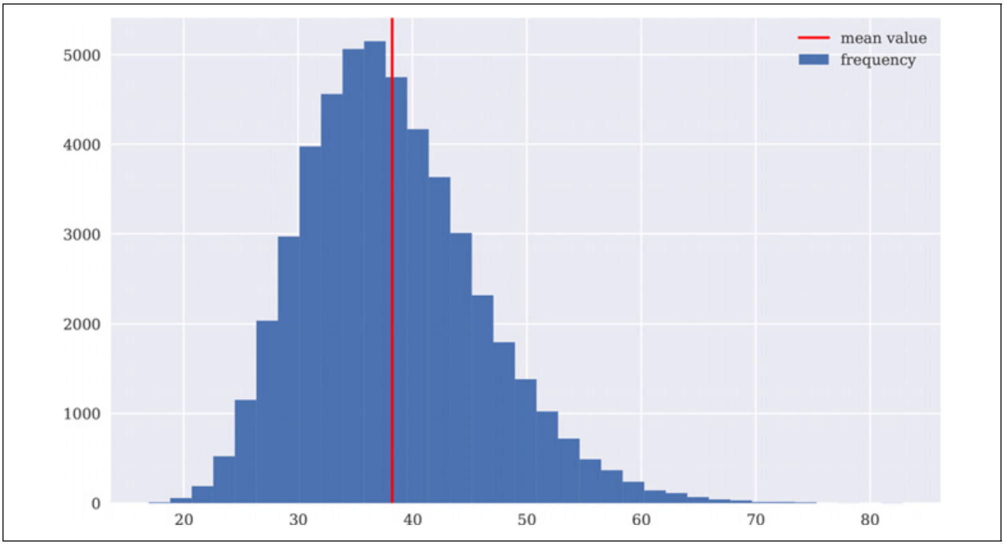


图 10-2　模拟期末价值的频率分布

10.4.2 NumPy

NumPy 版本的`mcs_simulation_np()`没有太多的不同，它仍然有一个 Python 循环，时间间隔是循环的一次执行。其他方面由所有路径上的向量化代码处理。它的速度比第一个版本大约快 20 倍：

```
In [127]: def mcs_simulation_np(p):
              M, I = p
              dt = T / M
              S = np.zeros((M + 1, I))
              S[0] = S0
              rn = np.random.standard_normal(S.shape)
              for t in range(1, M + 1): ❶
                  S[t] = S[t-1] * np.exp((r - sigma ** 2 / 2) * dt +
                                         sigma * math.sqrt(dt) * rn[t]) ❷
              return S

In [128]: %time S = mcs_simulation_np((M, I))
          CPU times: user 252 ms, sys: 32.9 ms, total: 285 ms
          Wall time: 252 ms

In [129]: S[-1].mean()
Out[129]: 38.235136032258595

In [130]: %timeit S = mcs_simulation_np((M, I))
          202 ms ± 27.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

❶ 在时间间隔上循环。

❷ 这个欧拉格式用向量化的 NumPy 代码一次性处理所有路径。

10.4.3 Numba

Numba 经常用于这类算法以取得显著的性能改善，这应该已经不再是令人惊讶的事了。Numba 版本的`mcs_simulation_nb()`略快于 NumPy 版本：

```
In [131]: mcs_simulation_nb = numba.jit(mcs_simulation_py)

In [132]: %time S = mcs_simulation_nb((M, I)) ❶
          CPU times: user 673 ms, sys: 36.7 ms, total: 709 ms
          Wall time: 764 ms

In [133]: %time S = mcs_simulation_nb((M, I)) ❷
          CPU times: user 239 ms, sys: 20.8 ms, total: 259 ms
          Wall time: 265 ms

In [134]: S[-1].mean()
```

```
Out[134]: 38.22350694016539

In [135]: C0 = math.exp(-r * T) * np.maximum(K - S[-1], 0).mean()

In [136]: C0
Out[136]: 3.8303077438193833

In [137]: %timeit S = mcs_simulation_nb((M, I))❷
          248 ms ± 20.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

❶ 第一次调用有编译开销。

❷ 第二次调用没有开销。

10.4.4 Cython

Cython 同样不出意料地需要花费更多精力，才能得到加速的效果。不过，加速并不明显。Cython 版本的 `mcs_simulation_cy()` 甚至稍微慢于 NumPy 和 Numba 版本。由于其他因素，将模拟结果转换为 `ndarray` 对象需要一些时间：

```
In [138]: %%cython
          import numpy as np
          cimport numpy as np
          cimport cython
          from libc.math cimport exp, sqrt
          cdef float S0 = 36.
          cdef float T = 1.0
          cdef float r = 0.06
          cdef float sigma = 0.2
          @cython.boundscheck(False)
          @cython.wraparound(False)
          def mcs_simulation_cy(p):
              cdef int M, I
              M, I = p
              cdef int t, i
              cdef float dt = T / M
              cdef double[:, :] S = np.zeros((M + 1, I))
              cdef double[:, :] rn = np.random.standard_normal((M + 1, I))
              S[0] = S0
              for t in range(1, M + 1):
                  for i in range(I):
                      S[t, i] = S[t-1, i] * exp((r - sigma ** 2 / 2) * dt +
                                                 sigma * sqrt(dt) * rn[t, i])
              return np.array(S)

In [139]: %time S = mcs_simulation_cy((M, I))
          CPU times: user 237 ms, sys: 65.2 ms, total: 302 ms
          Wall time: 271 ms
```

```
In [140]: S[-1].mean()
Out[140]: 38.241735841791574

In [141]: %timeit S = mcs_simulation_cy((M, I))
          221 ms ± 9.26 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

10.4.5 多进程

蒙特卡洛模拟是很适合并行化的任务。例如，方法之一是将 10 万条路径的模拟分为 10 个并行的进程，每个进程模拟 1 万条路径。另一种方法是将 10 万条路径的模拟分在多个进程中，每个进程模拟不同的金融工具。前者按照固定数量进程来并行模拟大量路径。下文的代码再次使用 multiprocessing 模块。它将全部模拟路径（I）分为较小的块 $\frac{I}{p}$（$p>0$）。所有单独任务完成后，结果通过 `np.hstack()`集中到一个 ndarray 对象。这种方法可应用于前面介绍的任何一个版本。

对这里选择的参数，并行方法没有观察到任何加速效果：

```
In [142]: import multiprocessing as mp

In [143]: pool = mp.Pool(processes=4) ❶

In [144]: p = 20 ❷

In [145]: %timeit S = np.hstack(pool.map(mcs_simulation_np,
                                         p * [(M, int(I / p))]))
          288 ms ± 10.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [146]: %timeit S = np.hstack(pool.map(mcs_simulation_nb,
                                         p * [(M, int(I / p))]))
          258 ms ± 8.69 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [147]: %timeit S = np.hstack(pool.map(mcs_simulation_cy,
                                         p * [(M, int(I / p))]))
          274 ms ± 11.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

❶ 用于并行化的 `Pool` 对象。

❷ 模拟分块的数量。

多进程策略

在金融业务中，有许多算法适用于并行化。其中一些算法甚至可以应用不同的代码并行化策略。蒙特卡洛模拟就是一个很好的例子，不管在单一机器或者多台机器上，都很容易并行执行多次模拟。这种算法本身还可以将单一模拟分布到多个进程上。

10.5　pandas 递归算法

本节介绍一个在金融分析中很重要的特殊主题：在 pandas 的 `DataFrame` 对象中存储的金融时间序列上实现递归函数。虽然 pandas 可以在 `DataFrame` 对象上实现复杂的向量化操作，但某些递归算法很难或者无法向量化，金融分析人员不得不在 `DataFrame` 对象上执行缓慢的 Python 循环。下面的例子以简单的形式实现了所谓的指数加权移动平均数（EWMA）。

金融时间序列 $S_t, t \in \{0, \cdots, T\}$的 EWMA 由公式 10-4 给出。

公式 10-4　指数加权移动平均数（EWMA）

$$EWMA_0 = S_0$$

$$EWMA_t = \alpha \cdot S_t + (1 - \alpha) \cdot EWMA_{t-1}, t \in \{1, \cdots, T\}$$

虽然性质很简单，也很容易实现，但这样的算法可能造成执行代码的速度相当慢。

10.5.1　Python

首先考虑 Python 版本，循环读取包含单一金融工具时间序列数据的 `DataFrame` 对象的 `DatetimeIndex`（参见第 8 章）。图 10-3 展示了这一金融时间序列和 EWMA 金融序列的图表：

```
In [148]: import pandas as pd

In [149]: sym = 'SPY'

In [150]: data = pd.DataFrame(pd.read_csv('../../source/tr_eikon_eod_data.csv',
                                index_col=0, parse_dates=True)[sym])
                                .dropna()
In [151]: alpha = 0.25

In [152]: data['EWMA'] = data[sym] ❶

In [153]: %%time
          for t in zip(data.index, data.index[1:]):
              data.loc[t[1], 'EWMA'] = (alpha * data.loc[t[1], sym] +
                                        (1 - alpha) * data.loc[t[0],
                                         'EWMA']) ❷
          CPU times: user 588 ms, sys: 16.4 ms, total: 605 ms
          Wall time: 591 ms

In [154]: data.head()
Out[154]:               SPY       EWMA
          Date
```

```
            2010-01-04 113.33 113.330000
            2010-01-05 113.63 113.405000
            2010-01-06 113.71 113.481250
            2010-01-07 114.19 113.658438
            2010-01-08 114.57 113.886328

In [155]: data[data.index > '2017-1-1'].plot(figsize=(10, 6));
```

❶ 初始化 EWMA 列。

❷ 以 Python 循环为基础，实现上述算法。

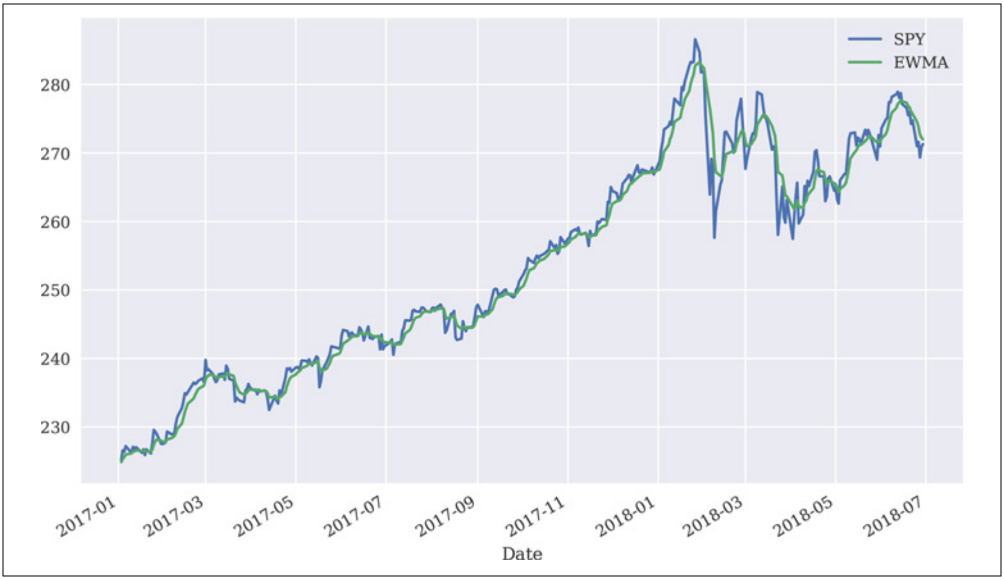


图 10-3　带 EWMA 的金融时间序列

现在考虑更为通用的 Python 函数 `ewma_py()`。该函数可以直接应用于 `ndarray` 对象形式的金融时间序列数据中的某列或者某行上：

```
In [156]: def ewma_py(x, alpha):
              y = np.zeros_like(x)
              y[0] = x[0]
              for i in range(1, len(x)):
                  y[i] = alpha * x[i] + (1-alpha) * y[i-1]
              return y

In [157]: %time data['EWMA_PY'] = ewma_py(data[sym], alpha) ❶
          CPU times: user 33.1 ms, sys: 1.22 ms, total: 34.3 ms
          Wall time: 33.9 ms
```

```
In [158]: %time data['EWMA_PY'] = ewma_py(data[sym].values, alpha) ❷
          CPU times: user 1.61 ms, sys: 44 µs, total: 1.65 ms
          Wall time: 1.62 ms
```

❶ 将函数直接应用到 `Series` 对象（列）。

❷ 将函数应用到包含行数据的 `ndarray` 对象。

这种方法已经显著加速了代码执行——大约 20 到 100 倍。

10.5.2 Numba

这个算法的基本结构有希望在应用 Numba 时进一步加速。确实如此，当 `ewma_nb()` 函数应用到 `ndarray` 数据时，再次加速了一个数量级：

```
In [159]: ewma_nb = numba.jit(ewma_py)

In [160]: %time data['EWMA_NB'] = ewma_nb(data[sym], alpha) ❶
          CPU times: user 269 ms, sys: 11.4 ms, total: 280 ms
          Wall time: 294 ms

In [161]: %timeit data['EWMA_NB'] = ewma_nb(data[sym], alpha) ❶
          30.9 ms ± 1.21 ms per loop (mean ± std. dev. of 7 runs, 10
          loops each)

In [162]: %time data['EWMA_NB'] = ewma_nb(data[sym].values, alpha) ❷
          CPU times: user 94.1 ms, sys: 3.78 ms, total: 97.9 ms
          Wall time: 97.6 ms

In [163]: %timeit data['EWMA_NB'] = ewma_nb(data[sym].values, alpha) ❷
          134 µs ± 12.5 µs per loop (mean ± std. dev. of 7 runs, 10000
          loops each)
```

❶ 函数直接应用 `Series` 对象（列）。

❷ 将函数应用到包含行数据的 `ndarray` 对象。

10.5.3 Cython

Cython 版本的 `ewma_cy()` 也实现了可观的速度改进，但不像 Numba 版本那么快：

```
In [164]: %%cython
          import numpy as np
          cimport cython
          @cython.boundscheck(False)
          @cython.wraparound(False)
          def ewma_cy(double[:] x, float alpha):
              cdef int i
```

```
                    cdef double[:] y = np.empty_like(x)
                    y[0] = x[0]
                    for i in range(1, len(x)):
                        y[i] = alpha * x[i] + (1 - alpha) * y[i - 1]
                    return y

In [165]: %time data['EWMA_CY'] = ewma_cy(data[sym].values, alpha)
          CPU times: user 2.98 ms, sys: 1.41 ms, total: 4.4 ms
          Wall time: 5.96 ms

In [166]: %timeit data['EWMA_CY'] = ewma_cy(data[sym].values, alpha)
          1.29 ms ± 194 µs per loop (mean ± std. dev. of 7 runs, 1000
          loops each)
```

最后这个例子再次说明，（非标准）算法的实现通常有多种选择。所有选择可以得出完全相同的结果，但性能特性上有很大的差别。本例的执行时间从 0.1ms 到 500ms 不等——相差 5000 倍。

最好和最优

将算法翻译成 Python 编程语言通常很容易。但是，由于性能上有很多种选择，所以算法的实现形式也很容易造成执行缓慢。对于交互式金融分析来说，最优的解决方案（也就是可以达到目的，但可能不是最快、也不是内存效率最高的）可能就足够了。对于生产型金融引用，应该坚持实现最好的解决方案，即使需要一些研究和正式基准测试都在所不惜。

10.6 结语

Python 生态系统提供了多种改善代码性能的手段。

风格与范型

在给定问题上，某些 Python 风格和范型比其他的性能更好；例如，在许多情况下，向量化范型不仅得到更简洁的代码，也能得到更高的速度（有时候以更大内存占用为代价）。

库

Python 有用于不同类型问题的大量库，使用适合于问题的库往往能得到性能更好的解决方案；NumPy 的 `ndarray` 类和 pandas 的 `DataFrame` 类就是很好的例子。

编译

有一些强大的编译解决方案能够加块金融算法的速度，如静态（如 Cython）和动态（例如 Numba）方案。

并行化

有些 Python 库（如 multiprocessing）可以轻松地并行化执行 Python 代码；本章的例子只使用了单一机器上的并行化，但 Python 生态系统还提供了多机器（群集）并行化的技术。

本章介绍的高性能方法的优点之一是：所有方法通常都很容易实现，而且需要的额外精力通常很少。换言之，在有这么多高性能库的今天，性能的改善是唾手可得的。

10.7 延伸阅读

本章介绍的所有高性能库，都有宝贵的在线资源：

- Cython 库及编译器项目的主页；
- multiprocessing 模块文档可以在 Python 官方文档上找到；
- 关于 Numba 的信息可以在 GitHub numba 页面和 Numba 官网上找到。

参考书如下：

- Gorelick, Misha, and Ian Ozsvald (2014). *High Performance Python*. Sebastopol, CA: O’Reilly.
- Smith, Kurt (2015). *Cython*. Sebastopol, CA: O’Reilly.

第 11 章 数学工具

数学家是现代化世界的“主人”。

——比尔·盖德

从 20 世纪 80 年代和 90 年代华尔街出现所谓的“火箭科学家”以来，金融学已经发展成为一种应用数学学科。虽然早期金融学研究论文中出现的数学表达式和公式不多，但是这些表达式和公式已经成为当今金融学论文的主要组成部分，其他部分则是解释它们的文字。

本章介绍对金融学有用的一些数学工具，但是对每种工具的背景不做详细的介绍。这个主题有许多实用的图书，因此，本章的焦点是如何在 Python 中使用这些工具和技术。这些工具如下所示。

逼近法

回归和插值是金融学中最常用的数学技术之一。

凸优化

一些金融学科需要凸优化工具（例如，衍生品定价模型检验）。

积分

金融（衍生品）资产的估值往往归结为积分计算。

符号数学

Python 提供 SymPy。SymPy 是一种强大的符号数学工具，例如，它可以解方程（组）。

11.1 逼近法

首先，是通常的导入工作：

```
In [1]: import numpy as np
        from pylab import plt, mpl

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

本节使用的主函数示例如下，由一个三角函数项和一个线性项组成：

```
In [3]: def f(x):
            return np.sin(x) + 0.5 * x
```

重点是在给定区间内通过回归和插值求取该函数的近似值。首先，生成该函数的图形，以便更好地观察逼近法的效果。我们感兴趣的区间是[−2π，2π]。图 11-1 显示了该函数在 `np.linspace()`函数定义的固定区间上的图像。`create_plot()`是一个助手函数，可以创建本章多次要使用的同类图表：

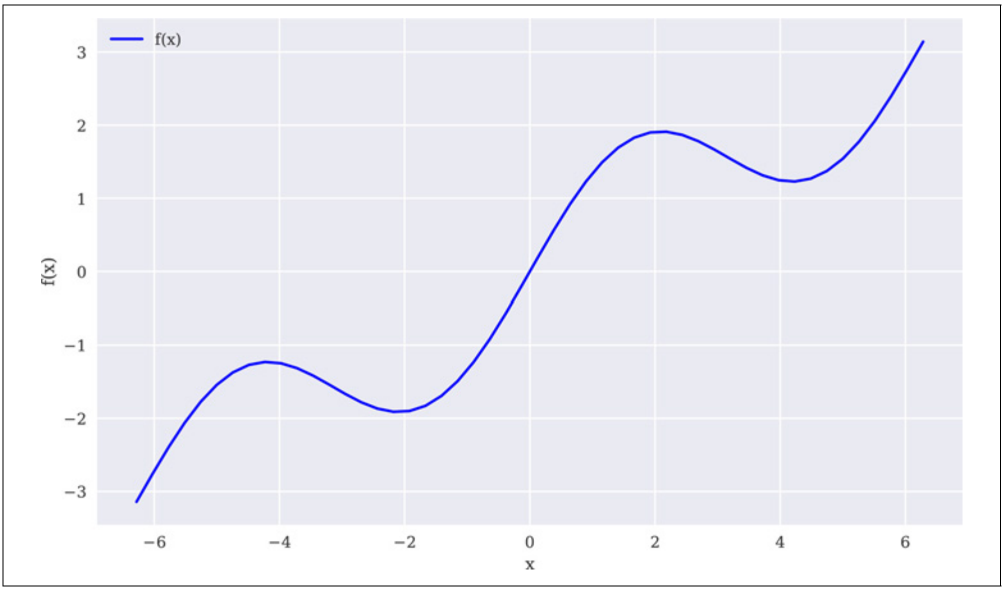


图 11-1　示例函数图表

```
In [4]: def create_plot(x, y, styles, labels, axlabels):
            plt.figure(figsize=(10, 6))
            for i in range(len(x)):
                plt.plot(x[i], y[i], styles[i], label=labels[i])
                plt.xlabel(axlabels[0])
                plt.ylabel(axlabels[1])
            plt.legend(loc=0)
```

```
In [5]: x = np.linspace(-2 * np.pi, 2 * np.pi, 50) ❶

In [6]: create_plot([x], [f(x)], ['b'], ['f(x)'], ['x', 'f(x)'])
```

❶ 用于绘图和计算的 x 值。

11.1.1 回归

回归是相当高效的函数近似值计算工具。它不仅适用于求取一维函数的近似值，在更高维度上也很有效。得出回归结果所需要的数值化方法很容易实现，执行也很快速。本质上，回归的任务是在给定一组所谓“基函数”b_d，$d \in \{1, \cdots, D\}$的情况下，根据公式 11-1 找出最优参数 α_1^*，⋯，α_D^*，其中对于 $i \in \{1, \cdots, I\}$观察点，$y_i \equiv f(x_i)$。x_i 可以视为自变量观测值，y_i 可视为因变量观测值（从函数或者统计的意义上说）。

公式 11-1　最小化回归问题

$$\min_{\alpha_1, \cdots, \alpha_D} \frac{1}{I} \sum_{i=1}^{I} [y_i - \sum_{d=1}^{D} \alpha_d \cdot b_d(x_i)]^2$$

1. 作为基函数的单项式

最简单的情况是以单项式作为基函数——也就是说，$b_1=1$，$b_2=x$，$b_3=x^2$，$b_4=x^3$⋯在这种情况下，NumPy 有可以确定最优参数（`np.polyfit()`）和通过一组输入值求取近似值（`np.polyval()`）的内建函数。

表 11-1 列出了 `np.polyfit()`函数的参数。在 `np.polyfit()`返回的最优回归相关系数 ρ 基础上，`np.polyval (ρ,x)`返回 x 坐标的回归值。

表 11-1　**`np.polyfit()`函数参数**

参数	描述
`x`	x 坐标（自变量值）
`y`	y 坐标（因变量值）
`deg`	多项式拟合度
`full`	如果为真，返回额外的诊断信息
`w`	应用到 y 坐标的权重
`cov`	如果为真，返回协方差矩阵

典型向量化风格的 `np.polyfit()`和 `np.polyval()`线性回归（`deg=1`）的应用方式如下。由于回归估算值保存在 `ry` 数组中，所以我们可以像图 11-2 那样比较回归结果和原始函数。当然，线性回归无法处理示例函数的 `sin` 部分：

```
In [7]: res = np.polyfit(x, f(x), deg=1, full=True) ❶

In [8]: res ❷
Out[8]: (array([ 4.28841952e-01, -1.31499950e-16]),
         array([21.03238686]),
         2,
         array([1., 1.]),
         1.1102230246251565e-14)

In [9]: ry = np.polyval(res[0], x) ❸

In [10]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ 线性回归步骤。

❷ 完整的结果：回归参数、残差、有效秩、奇异值和相对条件数。

❸ 使用回归参数求值。

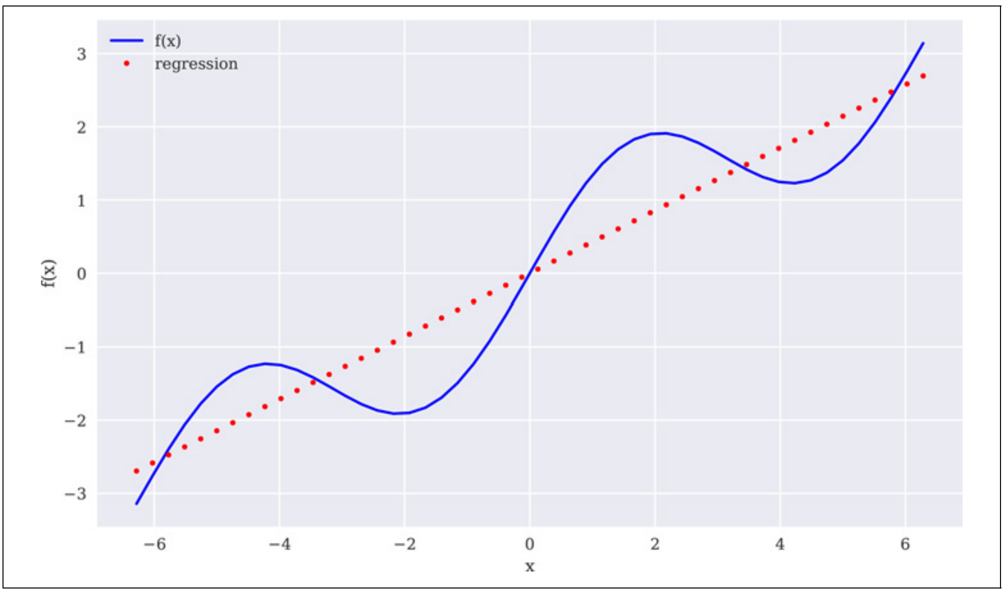


图 11-2　线性回归

为了处理示例函数的 sin 部分，必须使用更高次的单项式。下一个回归试图使用 5 次单项式作为基函数。果不其然，回归结果（如图 11-3 所示）看上去更接近原始函数。但是，它还远称不上完美：

```
In [11]: reg = np.polyfit(x, f(x), deg=5)
         ry = np.polyval(reg, x)
```

```
In [12]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

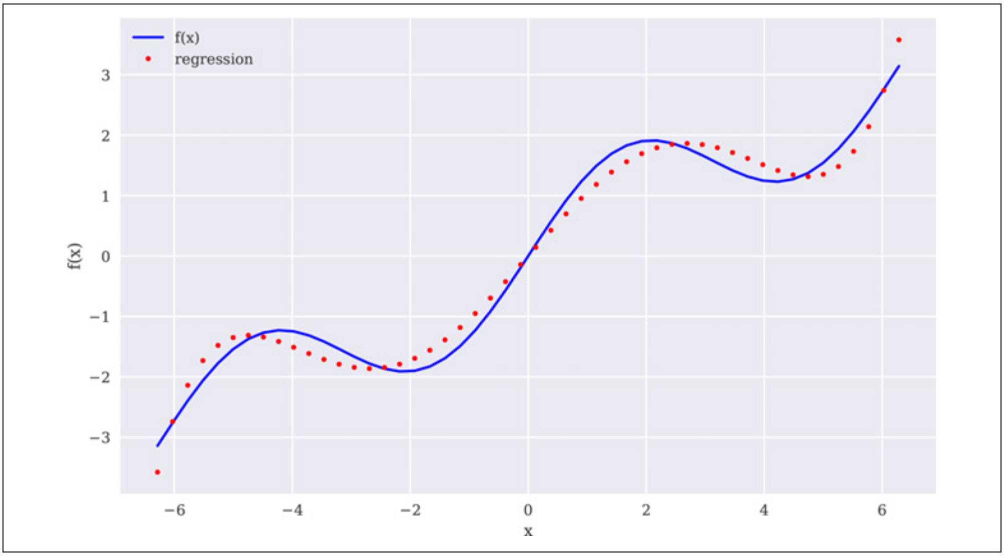


图 11-3　使用最高 5 次的单项式进行回归

最后一次尝试使用 7 次的单项式作为基函数来计算示例函数的近似值。这次的结果如图 11-4 所示，相当有说服力：

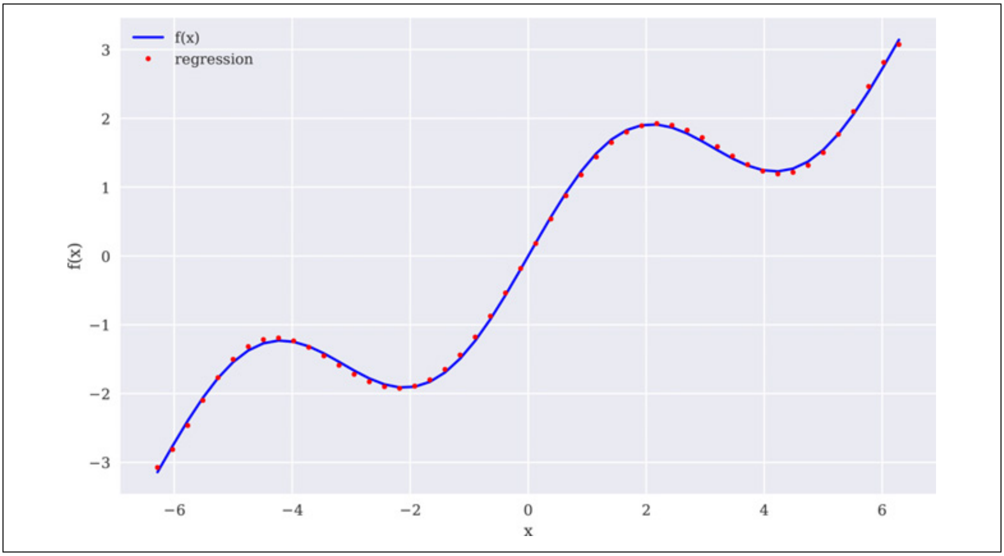


图 11-4　7 次单项式回归

```
In [13]: reg = np.polyfit(x, f(x), 7)
```

```
        ry = np.polyval(reg, x)

In [14]: np.allclose(f(x), ry)❶
Out[14]: False

In [15]: np.mean((f(x) - ry) ** 2) ❷
Out[15]: 0.0017769134759517689

In [16]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ 检查函数和回归值是否相同（至少接近）。

❷ 根据函数值计算回归值均方差（MSE）。

2. 单独的基函数

一般来说，当你选择更好的基函数组时，可以得到更好的回归结果，例如利用对函数的认识进行近似值计算。在这种情况下，单独的基函数必须通过一个矩阵方法定义（也就是使用 NumPy 的 `ndarray` 对象）。首先，例子中的多项式最高为 3 次（见图 11-5）。本例的核心函数是 `np.linalg.lstsq()`：

```
In [17]: matrix = np.zeros((3 + 1, len(x))) ❶
         matrix[3, :] = x ** 3 ❷
         matrix[2, :] = x ** 2 ❷
         matrix[1, :] = x ❷
         matrix[0, :] = 1 ❷

In [18]: reg = np.linalg.lstsq(matrix.T, f(x), rcond=None)[0] ❸

In [19]: reg.round(4) ❹
Out[19]: array([ 0. , 0.5628, -0. , -0.0054])

In [20]: ry = np.dot(reg, matrix) ❺

In [21]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ 基函数值（矩阵）所用的 `ndarray` 对象。

❷ 从常数到三次基函数值。

❸ 回归步骤。

❹ 最优回归参数。

❺ 函数值的回归估算。

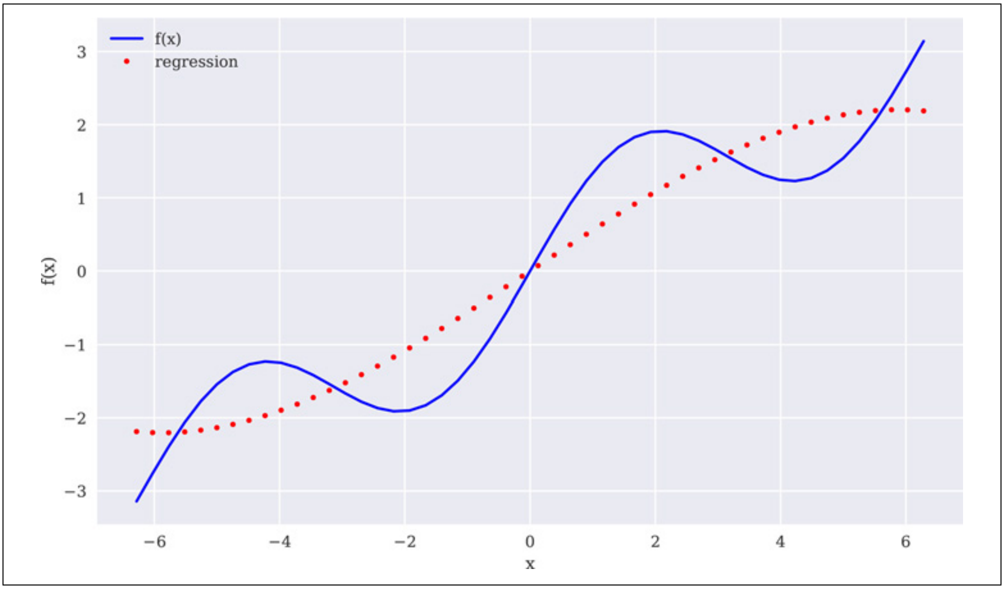


图 11-5　有单独基函数的回归

根据前面单项式的经验，图 11-5 中的结果并不如预期那么好。使用更通用的方法可以让我们利用对示例函数的认识。我们知道函数中有一个 sin 部分。因此，在基函数中包含一个正弦函数是有意义的。简单起见，我们替换最高次的单项式。现在的拟合很完美，如图 11-6 所示：

```
In [22]: matrix[3, :] = np.sin(x) ❶

In [23]: reg = np.linalg.lstsq(matrix.T, f(x), rcond=None)[0]

In [24]: reg.round(4) ❷
Out[24]: array([0. , 0.5, 0. , 1. ])

In [25]: ry = np.dot(reg, matrix)

In [26]: np.allclose(f(x), ry) ❸
Out[26]: True

In [27]: np.mean((f(x) - ry) ** 2) ❸
Out[27]: 3.404735992885531e-31

In [28]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ 新的基函数利用关于示例函数的知识。

❷ 最优回归参数恢复原始参数。

❸ 现在，回归产生了完美的拟合。

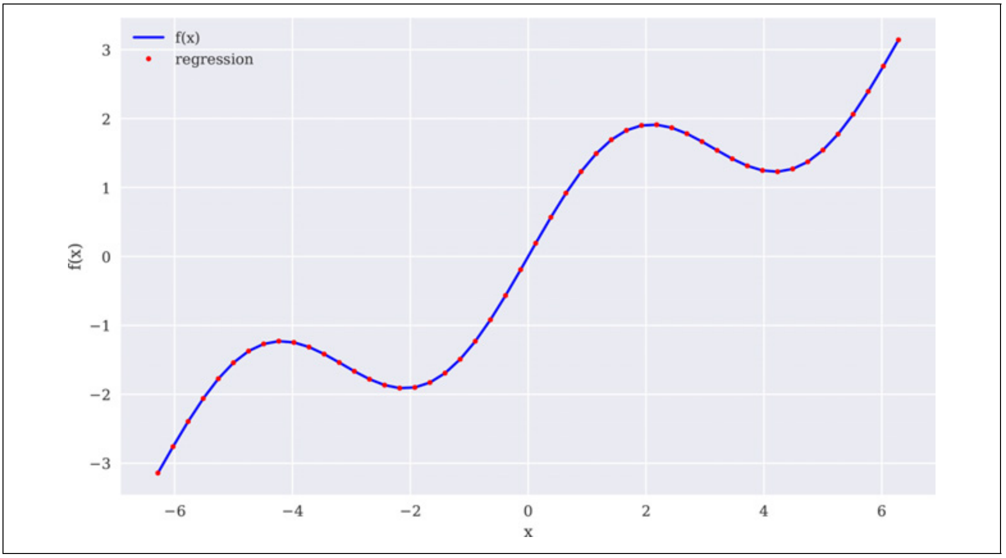


图 11-6　使用正弦基函数的回归

3. 有噪声的数据

对于有噪声的数据回归同样能够很好地处理，这种数据来自于模拟或者（不完善的）测量。为了阐述这个要点，我们生成同样具有噪声的自变量观测值和因变量观测值。图 11-7 表明，回归结果比有噪声的数据点更接近原始函数。在某种意义上，回归在一定程度上平均了噪声：

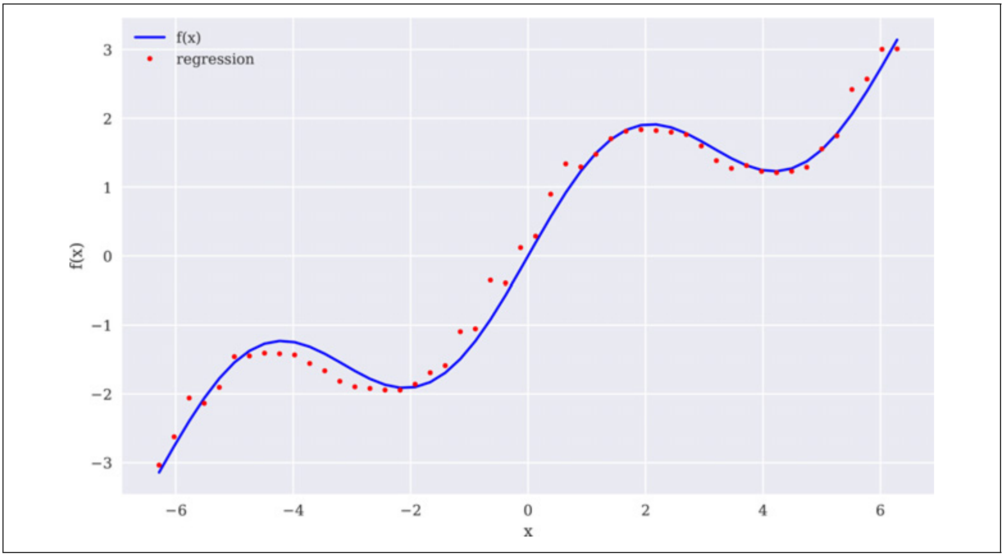


图 11-7　使用有噪声数据的回归

```
In [29]: xn = np.linspace(-2 * np.pi, 2 * np.pi, 50) ❶
         xn = xn + 0.15 * np.random.standard_normal(len(xn)) ❷
         yn = f(xn) + 0.25 * np.random.standard_normal(len(xn)) ❸

In [30]: reg = np.polyfit(xn, yn, 7)
         ry = np.polyval(reg, xn)
In [31]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ 新的 x 确定值。

❷ 在 x 值中引入噪声。

❸ 在 y 值中引入噪声。

4. 未排序数据

回归的另一个重要特点是，它可以无缝地处理未排序数据。前面的例子都依赖于经过排序的 x 数据，情况并不总是这样的。为了说明这一点，我们随机生成自变量数据点。在这种情况下，仅从视觉上检查原始数据很难识别出任何结构：

```
In [32]: xu = np.random.rand(50) * 4 * np.pi - 2 * np.pi ❶
         yu = f(xu)

In [33]: print(xu[:10].round(2)) ❶
         print(yu[:10].round(2)) ❶
         [-4.17 -0.11 -1.91 2.33 3.34 -0.96 5.81 4.92 -4.56 -5.42]
         [-1.23 -0.17  -1.9 1.89 1.47 -1.29 2.45 1.48 -1.29 -1.95]

In [34]: reg = np.polyfit(xu, yu, 5)
         ry = np.polyval(reg, xu)
In [35]: create_plot([xu, xu], [yu, ry], ['b.', 'ro'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ 随机化 x 值。

和有噪声数据一样，回归方法不关心观测点的顺序。这在研究公式 11-1 所示的最小化问题的结构时很明显。从图 11-8 中显示的结果来看也很明显。

5. 多维

最小二乘回归方法的另一个优点是，不需要太多的修改就可以用于多维的情况。接下来以 `fm()` 函数为例进行讲解：

```
In [36]: def fm(p):
             x, y = p
             return np.sin(x) + 0.25 * x + np.sqrt(y) + 0.05 * y ** 2
```

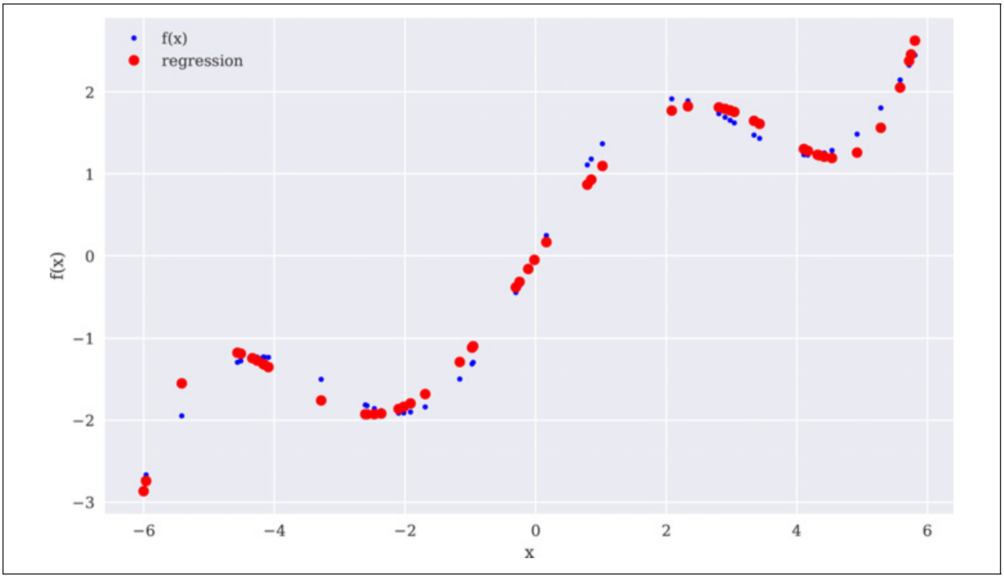


图 11-8　使用未排序数据的回归

为了正确地可视化这个函数，我们需要自变量数据点的网格（在两个维度上）。图 11-9 根据以 *x*、*y*、*z* 表示的自变量和因变量二维数据点网格，显示了 `fm()` 函数的形状：

```
In [37]: x = np.linspace(0, 10, 20)
         y = np.linspace(0, 10, 20)
         X, Y = np.meshgrid(x, y) ❶

In [38]: Z = fm((X, Y))
         x = X.flatten() ❷
         y = Y.flatten() ❷

In [39]: from mpl_toolkits.mplot3d import Axes3D ❸

In [40]: fig = plt.figure(figsize=(10, 6))
         ax = fig.gca(projection='3d')
         surf = ax.plot_surface(X, Y, Z, rstride=2, cstride=2,
                                cmap='coolwarm', linewidth=0.5,
                                antialiased=True)
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('f(x, y)')
         fig.colorbar(surf, shrink=0.5, aspect=5)
```

❶ 从一维 `ndarray` 对象生成二维 `ndarray` 对象（网格）。

❷ 从二维 `ndarray` 对象得到一维 `ndarray` 对象。

❸ 必要时从 `matplotlib` 导入 3D 绘图功能。

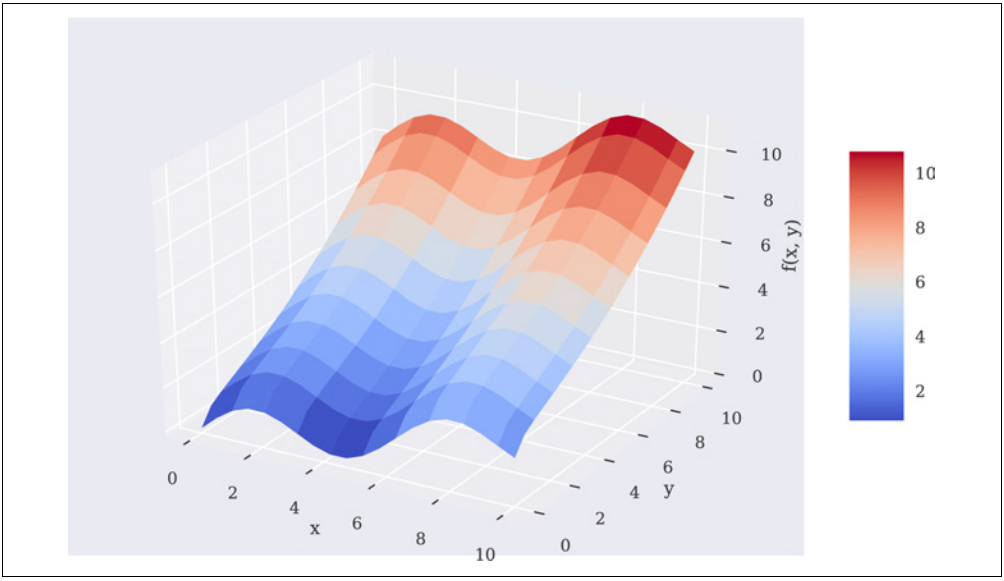


图 11-9　使用两个参数的函数

为了获得好的回归结果，我们将应用基本函数集，包括 fm() 函数、np.sin() 和 np.sqrt() 函数。图 11-10 直观展示了完美的回归结果：

```
In [41]: matrix = np.zeros((len(x), 6 + 1))
         matrix[:, 6] = np.sqrt(y) ❶
         matrix[:, 5] = np.sin(x) ❷
         matrix[:, 4] = y ** 2
         matrix[:, 3] = x ** 2
         matrix[:, 2] = y
         matrix[:, 1] = x
         matrix[:, 0] = 1

In [42]: reg = np.linalg.lstsq(matrix, fm((x, y)), rcond=None)[0]

In [43]: RZ = np.dot(matrix, reg).reshape((20, 20)) ❸

In [44]: fig = plt.figure(figsize=(10, 6))
         ax = fig.gca(projection='3d')
         surf1 = ax.plot_surface(X, Y, Z, rstride=2, cstride=2,
                     cmap=mpl.cm.coolwarm, linewidth=0.5,
                     antialiased=True) ❹
         surf2 = ax.plot_wireframe(X, Y, RZ, rstride=2, cstride=2,
                                   label='regression') ❺
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('f(x, y)')
         ax.legend()
         fig.colorbar(surf, shrink=0.5, aspect=5)
```

❶ 用于 *y* 参数的 `np.sqrt()`函数。

❷ 用于 *x* 参数的 `np.sin()`函数。

❸ 将回归结果转化为网格结构。

❹ 绘制原始函数曲面。

❺ 绘制回归曲面。

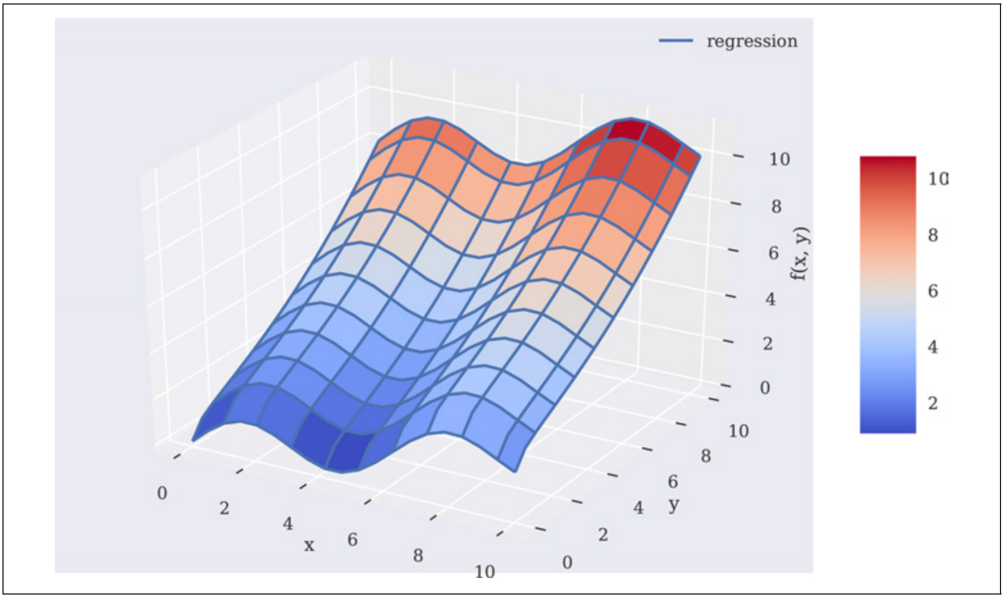


图 11-10　双参数函数的回归曲面

回归

最小二乘回归方法有多种应用领域，包括简单的函数逼近和基于有噪声或者未排序数据的函数逼近。这些方法可以应用于一维问题，也可以应用于多维问题。由于这种方法的基础数学理论，所以它在一维问题和多维问题上的应用总是“几乎相同”。

11.1.2　插值

与回归相比，插值（例如，3 次样条插值）在数学上更为复杂。它还被限制在低维度问题上。给定一组有序的观测点（按照 *x* 维排序），基本的思路是在两个相邻数据点之间进行回归，这样做不仅产生的分段插值函数完全匹配数据点，而且函数在数据点上连续可微分。连续可微分性需要至少 3 阶插值——也就是 3 次样条插值。然而，这种方法一般也适用于 4 次或者线性样条插值。

下面的代码可实现线性样条插值，结果如图 11-11 所示：

```
In [45]: import scipy.interpolate as spi ❶

In [46]: x = np.linspace(-2 * np.pi, 2 * np.pi, 25)

In [47]: def f(x):
             return np.sin(x) + 0.5 * x

In [48]: ipo = spi.splrep(x, f(x), k=1) ❷

In [49]: iy = spi.splev(x, ipo) ❸

In [50]: np.allclose(f(x), iy) ❹
Out[50]: True

In [51]: create_plot([x, x], [f(x), iy], ['b', 'ro'],
                     ['f(x)', 'interpolation'], ['x', 'f(x)'])
```

❶ 从 SciPy 导入必要的子库。

❷ 实现线性样条插值。

❸ 得出内插值。

❹ 检查内插值是否（足够）接近函数值。

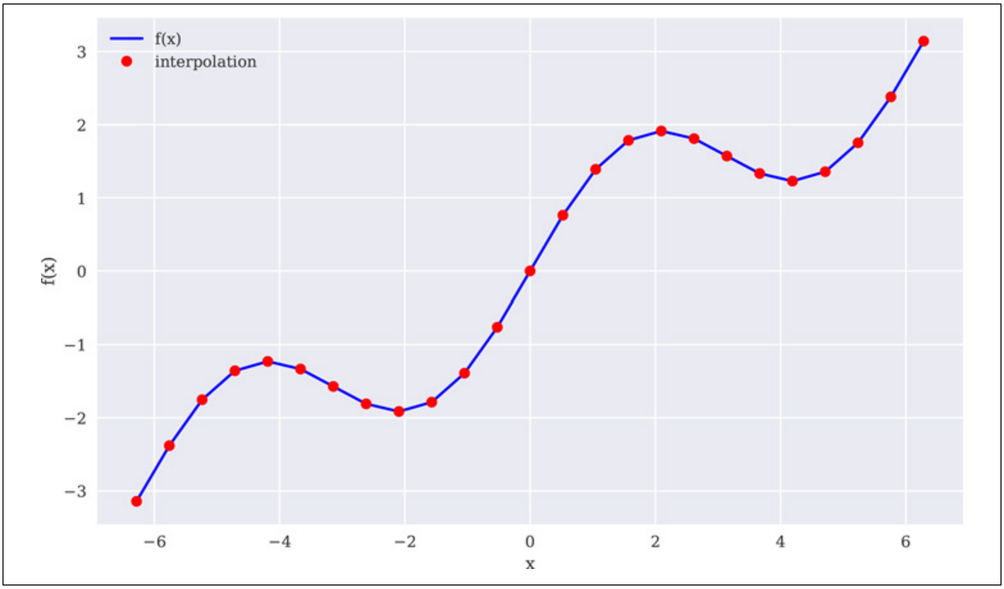


图 11-11　线性样条插值（完整数据集）

如果有按照 *x* 值排序的一组数据点，那么应用本身也和使用 `np.polyfit()`和 `np.polyval()`函数一样简单。在本例中，它们对应的函数是 `sci.splrep()`和 `sci.splev()`。表 11-2 列出了 `sci.splrep()`函数的主要参数。

表 11-2 sci.splrep()函数参数

参数	描述
x	（有序）*x* 坐标（自变量值）
y	（按 *x* 排序）*y* 坐标（因变量值）
w	应用到 *y* 坐标的权重
xb、xe	拟合区间，如果设置为 None 则区间为`[x[0], x[-1]]`
k	样条拟合顺序（$1 \leqslant k \leqslant 5$）
s	平滑因子（越大越平滑）
full_output	如果为真，返回附加输出
quiet	如果为真，抑制消息

表 11-3 列出 `sci.splev()`函数的参数。

表 11-3 sci.splev()函数参数

参数	描述
x	（有序）*x* 坐标（自变量值）
tck	`splrep()`返回的长度为 3 的序列（节点、系数、阶数）
der	导数的阶（`0` 为元函数，`1` 为一阶导数）
ext	*x* 不在节点序列中时的行为（`0` 外推，`1` 返回 0，`2` 引发 ValueError 异常）

样条插值在金融学中往往用于估算未包含在原始观测点中的自变量数据点的因变量值。为此，在下个例子中选择一个更小的区间，仔细观察一次样条插入的值。图 11-12 说明，插值函数确实可以线性地在两个观测点之间插值。对于某些应用，这可能不够精确。此外，很明显函数在原始数据点上不是连续可微分的——这是另一个不足：

```
In [52]: xd = np.linspace(1.0, 3.0, 50) ❶
         iyd = spi.splev(xd, ipo)

In [53]: create_plot([xd, xd], [f(xd), iyd], ['b', 'ro'],
                     ['f(x)', 'interpolation'], ['x', 'f(x)'])
```

❶ 具有更多数据点的较小区间。

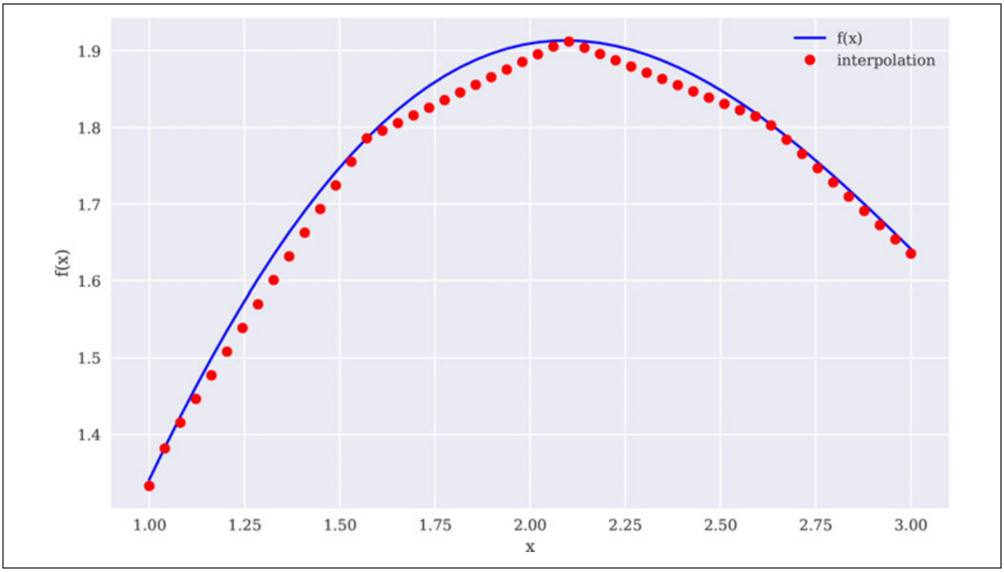


图 11-12　线性样条插值（数据子集）

重复整个练习，这次使用 3 次样条，结果明显改善（见图 11-13）：

```
In [54]: ipo = spi.splrep(x, f(x), k=3) ❶
         iyd = spi.splev(xd, ipo) ❷

In [55]: np.allclose(f(xd), iyd) ❸
Out[55]: False

In [56]: np.mean((f(xd) - iyd) ** 2) ❹
Out[56]: 1.1349319851436892e-08

In [57]: create_plot([xd, xd], [f(xd), iyd], ['b', 'ro'],
                     ['f(x)', 'interpolation'], ['x', 'f(x)'])
```

❶ 完整数据集上的 3 次样条插值。

❷ 结果应用到更小的时间间隔。

❸ 插值仍然不完美。

❹ 但好于从前。



插值

在可以应用样条插值的情况下，可以期望得到比最小二乘回归方法更好的近似结果。但是要记住，必须有排序（且“无噪声”）的数据，该方法仅限于低维度问题。样条插值的计算要求也更高，在某些用例中可能导致花费的时间比回归方法的长得多。

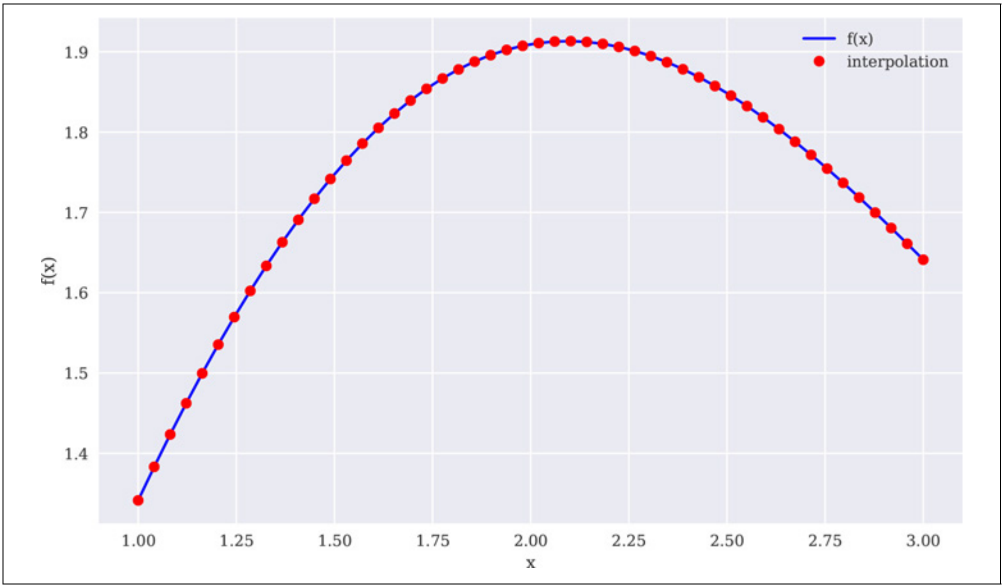


图 11-13　3 次样条插值（数据子集）

11.2　凸优化

在金融学和经济学中，凸优化起着重要的作用。这方面的例子包括根据市场数据校准期权定价模型，或者代理人效用函数的优化。我们以下面定义的函数 fm()为例进行这种优化：

```
In [58]: def fm(p):
             x, y = p
             return (np.sin(x) + 0.05 * x ** 2
                   + np.sin(y) + 0.05 * y ** 2)
```

图 11-14 展示了上述函数在定义的 x 和 y 区间内的图形。从图 11-14 可以看出这个函数有多个局部极小值。从这个特定的图形表现中无法真正确认全局最小值是否存在：

```
In [59]: x = np.linspace(-10, 10, 50)
         y = np.linspace(-10, 10, 50)
         X, Y = np.meshgrid(x, y)
         Z = fm((X, Y))

In [60]: fig = plt.figure(figsize=(10, 6))
         ax = fig.gca(projection='3d')
         surf = ax.plot_surface(X, Y, Z, rstride=2, cstride=2,
```

```
                                    cmap='coolwarm', linewidth=0.5,
                                    antialiased=True)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
fig.colorbar(surf, shrink=0.5, aspect=5)
```

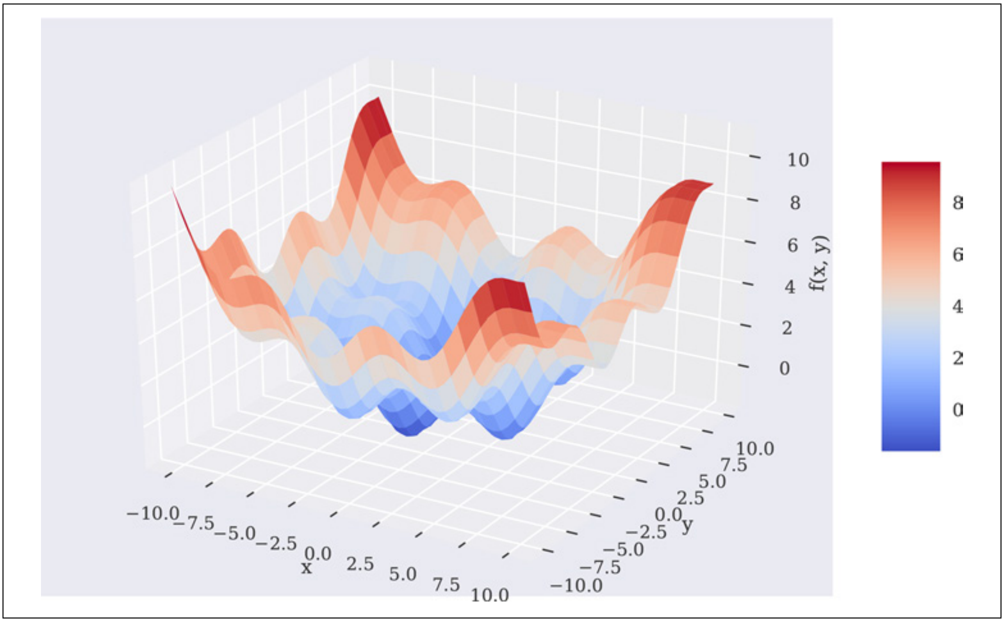


图 11-14 以两个参数最小化的函数

11.2.1 全局优化

下面将实现全局最小化和局部最小化。所应用的 `sco.brute()`和 `sco.fmin()`函数来自 `scipy.optimize()`。

为了仔细观察这些最小过程的幕后操作，使用以下代码为原函数增补可以输出当前参数值和函数值的选项。这使我们可以跟踪过程的所有相关信息：

```
In [61]: import scipy.optimize as sco ❶

In [62]: def fo(p):
             x, y = p
             z = np.sin(x) + 0.05 * x ** 2 + np.sin(y) + 0.05 * y ** 2
             if output == True:
                 print('%8.4f | %8.4f | %8.4f' % (x, y, z)) ❷
             return z
```

```
In [63]: output = True
         sco.brute(fo, ((-10, 10.1, 5), (-10, 10.1, 5)), finish=None) ❸
         -10.0000  | -10.0000 | 11.0880
         -10.0000  | -10.0000 | 11.0880
         -10.0000  |  -5.0000 |  7.7529
         -10.0000  |   0.0000 |  5.5440
         -10.0000  |   5.0000 |  5.8351
         -10.0000  |  10.0000 | 10.0000
          -5.0000  | -10.0000 |  7.7529
          -5.0000  |  -5.0000 |  4.4178
          -5.0000  |   0.0000 |  2.2089
          -5.0000  |   5.0000 |  2.5000
          -5.0000  |  10.0000 |  6.6649
           0.0000  | -10.0000 |  5.5440
           0.0000  |  -5.0000 |  2.2089
           0.0000  |   0.0000 |  0.0000
           0.0000  |   5.0000 |  0.2911
           0.0000  |  10.0000 |  4.4560
           5.0000  | -10.0000 |  5.8351
           5.0000  |  -5.0000 |  2.5000
           5.0000  |   0.0000 |  0.2911
           5.0000  |   5.0000 |  0.5822
           5.0000  |  10.0000 |  4.7471
          10.0000  | -10.0000 | 10.0000
          10.0000  |  -5.0000 |  6.6649
          10.0000  |   0.0000 |  4.4560
          10.0000  |   5.0000 |  4.7471
          10.0000  |  10.0000 |  8.9120

Out[63]: array([0., 0.])
```

❶ 从 SciPy 导入必要的子库。

❷ 如果 `output = True`，打印信息。

❸ 暴力优化。

根据函数的初始参数，最优参数值为 `x=y=0`。快速检查上述输出可以看出，结果函数值也为 0。你可能倾向于接受这一结果为全局优化。然而，第一次参数化的过程相当粗糙：对两个输入参数均使用 5 的步长。现在进行显著的细化，以得到更好的结果，同时也说明上述解决方案不是最优的：

```
In [64]: output = False
         opt1 = sco.brute(fo, ((-10, 10.1, 0.1), (-10, 10.1, 0.1)),
         finish=None)
```

```
In [65]: opt1
Out[65]: array([-1.4, -1.4])

In [66]: fm(opt1)
Out[66]: -1.7748994599769203
```

最优参数值现在为 `x=y=-1.4`，全局优化的最小函数值大约为−1.7749。

11.2.2 局部优化

接下来的局部凸优化需要用到全局优化的结果。`sco.fmin()`函数的输入是需要最小化的函数和起始参数值。可选的参数值是输入参数宽容度和函数值宽容度，以及最大迭代及函数调用次数。局部优化可进一步改善结果：

```
In [67]: output = True
         opt2 = sco.fmin(fo, opt1, xtol=0.001, ftol=0.001,
                         maxiter=15, maxfun=20) ❶
          -1.4000 | -1.4000 | -1.7749
          -1.4700 | -1.4000 | -1.7743
          -1.4000 | -1.4700 | -1.7743
          -1.3300 | -1.4700 | -1.7696
          -1.4350 | -1.4175 | -1.7756
          -1.4350 | -1.3475 | -1.7722
          -1.4088 | -1.4394 | -1.7755
          -1.4438 | -1.4569 | -1.7751
          -1.4328 | -1.4427 | -1.7756
          -1.4591 | -1.4208 | -1.7752
          -1.4213 | -1.4347 | -1.7757
          -1.4235 | -1.4096 | -1.7755
          -1.4305 | -1.4344 | -1.7757
          -1.4168 | -1.4516 | -1.7753
          -1.4305 | -1.4260 | -1.7757
          -1.4396 | -1.4257 | -1.7756
          -1.4259 | -1.4325 | -1.7757
          -1.4259 | -1.4241 | -1.7757
          -1.4304 | -1.4177 | -1.7757
          -1.4270 | -1.4288 | -1.7757
          Warning: Maximum number of function evaluations has been exceeded.

In [68]: opt2
Out[68]: array([-1.42702972, -1.42876755])

In [69]: fm(opt2)
Out[69]: -1.7757246992239009
```

❶ 局部凸优化。

在许多凸优化问题中，建议在局部优化之前进行全局优化。主要原因是局部凸优化算法很容易陷入某个局部最小值（所谓的“盆地跳跃”（basin hopping）)，而忽略“更好”的局部最小值和全局最小值。下面可以看到，将初始参数化设置为 x=y=2 可得出高于 0 的“最小”值：

```
In [70]: output = False
         sco.fmin(fo, (2.0, 2.0), maxiter=250)
         Optimization terminated successfully.
                  Current function value: 0.015826
                  Iterations: 46
                  Function evaluations: 86

Out[70]: array([4.2710728 , 4.27106945])
```

11.2.3 有约束优化

到目前为止，我们只考虑了无约束优化问题，但是许多类型的经济学或者金融学优化问题都有一个或者多个约束条件。这些约束可能采取等式或者不等式等正规形式。

举个简单的例子，考虑可能投资两种高风险证券（希望效用最大化）的投资者的效用最大化问题。两种证券今天的价格为 $q_a=q_b=10$ 美元。一年之后，状态 u 下它们的收益分别为 15 美元和 5 美元，而在状态 d 下收益分别为 5 美元和 12 美元。两种状态出现的可能性相同。两种证券的向量收益分别记为 r_a 和 r_b。

假设投资者的投资预算为 $w_0=100$ 美元，根据效用函数 $u(w)=\sqrt{w}$ 得出未来的财富效率，其中 w 是可用的财富（以美元计算）。公式 11-2 是最优化问题的公式，其中 a、b 是投资者购买的证券数量。

公式 11-2　预期效用最大化问题（1）

$$
\begin{aligned}
&\max_{a,b} E(u(w_1)) = p\sqrt{w_{1u}} + (1-p)\sqrt{w_{1d}} \\
&w_1 = a \bullet r_a + b \bullet r_b \\
&w_0 \geqslant a \bullet q_a + b \bullet q_b \\
&a,b \geqslant 0
\end{aligned}
$$

代入所有数值化假设，就得到公式 11-3。注意，我们还对负数的预期效用最小化做了更改。

公式 11-3　预期效用最大化问题（2）

$$
\begin{aligned}
&\min_{a,b} -E(u(w_1)) = -(0.5 \cdot \sqrt{w_{1u}} + 0.5 \cdot \sqrt{w_{1d}}) \\
&w_{1u} = a \cdot 15 + b \cdot 5 \\
&w_{1d} = a \cdot 5 + b \cdot 12 \\
&100 \geqslant a \cdot 10 + b \cdot 10 \\
&a,b \geqslant 0
\end{aligned}
$$

我们使用 `scipy.optimize.minimize()`函数来解决上述问题。除了需要最小化的函数之外，这个函数还将公式、不等式（以字典对象列表的形式）和参数范围（以元组对象元组的形式）作为输入。我们可以将公式 11-3 通过如下代码实现：

```
In [71]: import math

In [72]: def Eu(p): ❶
             s, b = p
             return -(0.5 * math.sqrt(s * 15 + b * 5) +
                      0.5 * math.sqrt(s * 5 + b * 12))

In [73]: cons = ({'type': 'ineq',
                  'fun': lambda p: 100 - p[0] * 10 - p[1] * 10}) ❷

In [74]: bnds = ((0, 1000), (0, 1000)) ❸

In [75]: result = sco.minimize(Eu, [5, 5], method='SLSQP',
                               bounds=bnds, constraints=cons) ❹
```

❶ 为了最大化预期效用而最小化的函数。

❷ 字典对象形式的不等式约束。

❸ 参数边界值（选择为足够宽）。

❹ 约束优化。

`result` 对象包含所有相关信息。对于最小函数值，必须记得恢复符号：

```
In [76]: result
Out[76]:      fun: -9.700883611487832
              jac: array([-0.48508096, -0.48489535])
          message: 'Optimization terminated successfully.'
             nfev: 21
              nit: 5
             njev: 5
           status: 0
          success: True
                x: array([8.02547122, 1.97452878])

In [77]: result['x'] ❶
Out[77]: array([8.02547122, 1.97452878])

In [78]: -result['fun'] ❷
Out[78]: 9.700883611487832

In [79]: np.dot(result['x'], [10, 10]) ❸
```

```
Out[79]: 99.99999999999999
```

❶ 最优参数值（即最优投资组合）。

❷ 函数的最小负数值是最优解决方案值。

❸ 预算约束是有约束力的；投入所有财富。

11.3　积分

在估值和期权定价时，积分是一个重要的数学工具。这来源于一个事实：衍生物的风险中立价值一般可以用风险中立（鞅）测度下的预期折现收益来表示。这一预期在离散情况下是个总和，在连续情况下是一个积分。`scipy.integrate` 子库提供了数值积分所用的不同函数。示例函数在 11.1 节中已经讲解过：

```
In [80]: import scipy.integrate as sci

In [81]: def f(x):
             return np.sin(x) + 0.5 * x
```

积分区间应该为`[0.5，9.5]`，这就得出了公式 11-4 所示的定积分。

公式 11-4　示例函数的积分

$$\int_{0.5}^{9.5} f(x)\mathrm{d}x = \int_{0.5}^{9.5} \sin(x) + 0.5x\mathrm{d}x$$

下面的代码定义了求取积分的主要 Python 对象：

```
In [82]: x = np.linspace(0, 10)
         y = f(x)
         a = 0.5 ❶
         b = 9.5 ❷
         Ix = np.linspace(a, b) ❸
         Iy = f(Ix) ❹
```

❶ 积分左界。

❷ 积分右界。

❸ 积分区间值。

❹ 积分函数值。

图 11-15 将积分值表示为函数下的灰色阴影区域：[1]

1 关于此类图表的讨论参见第 7 章。——原注

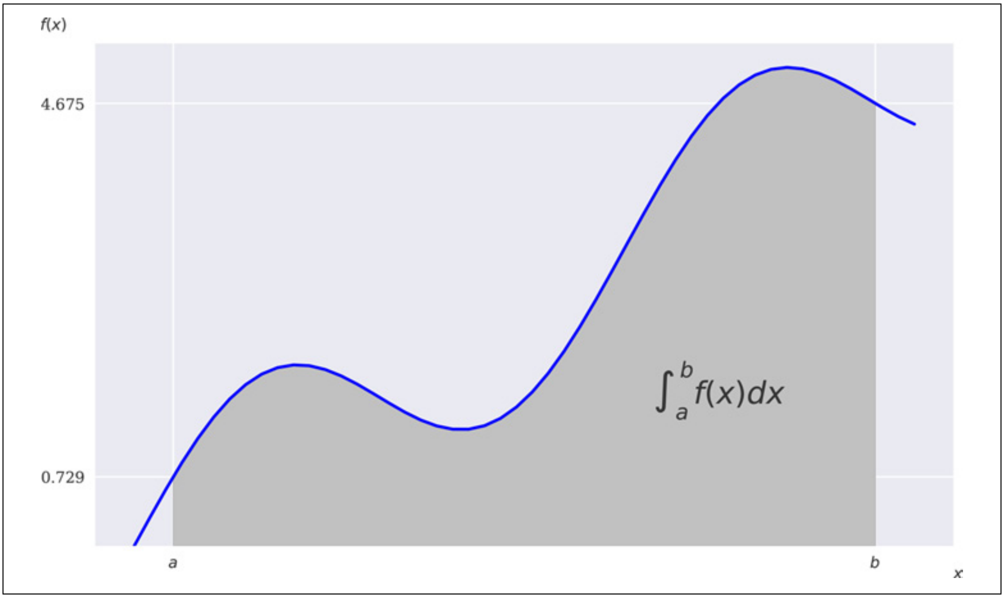


图 11-15　积分值表现为阴影面积

```
In [83]: from matplotlib.patches import Polygon

In [84]: fig, ax = plt.subplots(figsize=(10, 6))
         plt.plot(x, y, 'b', linewidth=2)
         plt.ylim(bottom=0)
         Ix = np.linspace(a, b)
         Iy = f(Ix)
         verts = [(a, 0)] + list(zip(Ix, Iy)) + [(b, 0)]
         poly = Polygon(verts, facecolor='0.7', edgecolor='0.5')
         ax.add_patch(poly)
         plt.text(0.75 * (a + b), 1.5, r"$\int_a^b f(x)dx$",
                  horizontalalignment='center', fontsize=20)
         plt.figtext(0.9, 0.075, '$x$')
         plt.figtext(0.075, 0.9, '$f(x)$')
         ax.set_xticks((a, b))
         ax.set_xticklabels(('$a$', '$b$'))
         ax.set_yticks([f(a), f(b)]);
```

11.3.1　数值积分

`scipy.integrate` 子库包含一组精选的函数，可以计算给定上下限和数学函数下的数值积分。这些函数的例子包含用于固定高斯求积的 `sci.fixed_quad()`、用于自适应求积的 `sci.quad()` 和用于龙贝格积分的 `sci.romberg()`：

```
In [85]: sci.fixed_quad(f, a, b)[0]
Out[85]: 24.366995967084602

In [86]: sci.quad(f, a, b)[0]
Out[86]: 24.374754718086752

In [87]: sci.romberg(f, a, b)
Out[87]: 24.374754718086713
```

还有一些积分函数以输入列表或者包含函数值和输入值的 `ndarray` 对象作为输入。这种函数的例子包括使用梯形法则的 `sci.trapz()`和实现辛普森法则的 `sci.simps()`：

```
In [88]: xi = np.linspace(0.5, 9.5, 25)

In [89]: sci.trapz(f(xi), xi)
Out[89]: 24.352733271544516

In [90]: sci.simps(f(xi), xi)
Out[90]: 24.37496418455075
```

11.3.2 通过模拟求取积分

通过蒙特卡洛模拟（参见第 12 章）的期权和衍生物估值基于这样一个认识——可以通过模拟来求取积分。为此，在积分区间内取 I 个随机的 `x` 值，并计算每个随机 `x` 值处的积分函数值。加总所有函数值并求其平均值，就可以得到积分区间的平均函数值。将该值乘以积分区间长度，可以得出估算的积分值。

下面的代码说明蒙特卡洛估算积分值如何随着提取随机数个数的增加而收敛（但并非单调收敛）。即使提取的随机数个数较少，估算值也已经相当接近：

```
In [91]: for i in range(1, 20):
             np.random.seed(1000)
             x = np.random.random(i * 10) * (b - a) + a ❶
             print(np.mean(f(x)) * (b - a))
         24.804762279331463
         26.522918898332378
         26.265547519223976
         26.02770339943824
         24.99954181440844
         23.881810141621663
         23.527912274843253
         23.507857658961207
         23.67236746066989
         23.679410416062886
         24.424401707879305
```

```
24.239005346819056
24.115396924962802
24.424191987566726
23.924933080533783
24.19484212027875
24.117348378249833
24.100690929662274
23.76905109847816
```

❶ 随机数 x 的值随每次循环增加。

11.4　符号计算

前几节主要关心的是数值计算。本节介绍符号计算，这种计算在许多金融领域中很有用处。我们通常使用专用于符号计算的 SymPy 库。

11.4.1　基础知识

SymPy 引入了新的对象类。最基本的是 Symbol 类：

```
In [92]: import sympy as sy

In [93]: x = sy.Symbol('x') ❶
         y = sy.Symbol('y') ❶

In [94]: type(x)
Out[94]: sympy.core.symbol.Symbol

In [95]: sy.sqrt(x) ❷
Out[95]: sqrt(x)

In [96]: 3 + sy.sqrt(x) - 4 ** 2 ❸
Out[96]: sqrt(x) – 13

In [97]: f = x ** 2 + 3 + 0.5 * x ** 2 + 3 / 2 ❹

In [98]: sy.simplify(f) ❺
Out[98]: 1.5*x**2 + 4.5
```

❶ 定义所要处理的符号。

❷ 对一个符号应用函数。

❸ 在符号上定义的数值表达式。

❹ 符号化定义的函数。

❺ 简化的函数表达式。

已经表现出了一个重大差异。尽管 x 没有数字值，但是 SymPy 已经定义了 x 的平方根，因为 x 是一个符号（Symbol）对象。在这个意义上，sy.sqrt(x)可以成为任何数学表达式的一部分。注意，SymPy 通常自动简化给定的数学表达式。我们可以用符号对象定义任何函数。它们不会和 Python 函数混淆。

SymPy 为数学表达式提供了 3 个基本的渲染器：

- 基于 LaTeX 的渲染器；
- 基于 Unicode 的渲染器；
- 基于 ASCII 的渲染器。

例如，若只在 Jupyter Notebook 环境（基于 HTML）中工作时，LaTex 渲染通常是好（在视觉上引人入胜）的选择。下面，我们将一直使用最简单的 ASCII 渲染器以表明不涉及任何人工制作的排版设定：

```
In [99]: sy.init_printing(pretty_print=False, use_unicode=False)

In [100]: print(sy.pretty(f))
               2
          1.5*x + 4.5

In [101]: print(sy.pretty(sy.sqrt(x) + 0.5))
            ___
          \/ x + 0.5
```

这里我们不过多地纠缠细节，SymPy 还提供了许多其他有用的数学函数——例如，用于求 π 值的函数。下面展示 π 的字符串表示的前 40 个字符（最多可达 40 万位）。这个程序还在 π 值中搜索 6 位生日（日期优先）——这是某些数学和 IT 圈子中流行的任务：

```
In [102]: %time pi_str = str(sy.N(sy.pi, 400000)) ❶
          CPU times: user 400 ms, sys: 10.9 ms, total: 411 ms
          Wall time: 501 ms

In [103]: pi_str[:42] ❷
Out[103]: '3.1415926535897932384626433832795028841971'

In [104]: pi_str[-40:] ❸
Out[104]: '8245672736856312185020980470362464176198'

In [105]: %time pi_str.find('061072') ❹
          CPU times: user 115 µs, sys: 1e+03 ns, total: 116 µs
```

```
        Wall time: 120 µs

Out[105]: 80847
```

❶ 返回 π 前 40 万位数字的字符串表示。

❷ 显示前 40 位。

❸ 显示最后 40 位。

❹ 在这个字符串中搜索一个生日。

11.4.2 方程式

SymPy 的长处之一是解方程，例如 x^2–1=0 这样的形式。一般来说，SymPy 假定你寻找的是使指定表达式为 0 的方程解。因此，x^2–1=3 这样的方程可能必须改写，才能获得想要的结果。当然，SymPy 可以处理更复杂的表达式，如 $x^3 + 0.5x^2 - 1 = 0$。最后，它还能解决涉及虚数的问题，如求 $x^2 + y^2 = 0$ 的解：

```
In [106]: sy.solve(x ** 2 - 1)
Out[106]: [-1, 1]

In [107]: sy.solve(x ** 2 - 1 - 3)
Out[107]: [-2, 2]

In [108]: sy.solve(x ** 3 + 0.5 * x ** 2 - 1)
Out[108]: [0.858094329496553, -0.679047164748276 - 0.839206763026694*I,
           -0.679047164748276 + 0.839206763026694*I]

In [109]: sy.solve(x ** 2 + y ** 2)
Out[109]: [{x: -I*y}, {x: I*y}]
```

11.4.3 积分与微分

SymPy 的另一个长处是积分和微分。下面，回到用于数值和模拟积分的示例函数，并既求出符号解，也求出精确的数值解。我们需要积分上下限的符号对象：

```
In [110]: a, b = sy.symbols('a b') ❶

In [111]: I = sy.Integral(sy.sin(x) + 0.5 * x, (x, a, b)) ❷

In [112]: print(sy.pretty(I)) ❷
            b
            /
           |
           | (0.5*x + sin(x)) dx
           |
```

```
          /
          a

In [113]: int_func = sy.integrate(sy.sin(x) + 0.5 * x, x) ❸

In [114]: print(sy.pretty(int_func)) ❸
                2
          0.25*x  - cos(x)

In [115]: Fb = int_func.subs(x, 9.5).evalf() ❹
          Fa = int_func.subs(x, 0.5).evalf() ❹

In [116]: Fb – Fa ❺
Out[116]: 24.3747547180867
```

❶ 积分区间的符号对象。

❷ 定义 `Integral` 对象并以漂亮的格式打印。

❸ 得出反导数并以漂亮的格式打印。

❹ 区间两端的反导数值，方法为`.subs()`和`.evalf()`函数。

❺ 准确的积分数值。

也可以用符号积分上下限求得积分符号解：

```
In [117]: int_func_limits = sy.integrate(sy.sin(x) + 0.5 * x, (x, a, b)) ❶

In [118]: print(sy.pretty(int_func_limits)) ❶
                  2          2
          - 0.25*a  + 0.25*b  + cos(a) - cos(b)

In [119]: int_func_limits.subs({a : 0.5, b : 9.5}).evalf() ❷
Out[119]: 24.3747547180868

In [120]: sy.integrate(sy.sin(x) + 0.5 * x, (x, 0.5, 9.5)) ❸
Out[120]: 24.3747547180867
```

❶ 求得积分符号解。

❷ 在替换中使用字典对象求得积分数值解。

❸ 只用一步求得积分数值解。

11.4.4 微分

对不定积分求导通常会得出原函数。我们对前面的符号反导数应用 `sy.diff()`函数来

验证这一点：

```
In [121]: int_func.diff()
Out[121]: 0.5*x + sin(x)
```

和积分示例一样，现在使用微分得出前面的凸最小化问题的解。为此，我们定义对应的符号函数，得出偏导数，并确定根。

全局最小值的必要（但不充分）条件之一是两个偏微分都为 0。但是，这不能保证有符号解。算法和（多种）存在性问题在这里都起了作用。然而，我们可以从数值上求出两个方程式的解，再根据前面的全局和局部最小化工作提供“有根据”的推测：

```
In [122]: f = (sy.sin(x) + 0.05 * x ** 2
             + sy.sin(y) + 0.05 * y ** 2) ❶

In [123]: del_x = sy.diff(f, x) ❷
          del_x ❷
Out[123]: 0.1*x + cos(x)

In [124]: del_y = sy.diff(f, y) ❷
          del_y ❷
Out[124]: 0.1*y + cos(y)

In [125]: xo = sy.nsolve(del_x, -1.5) ❸
          xo ❸
Out[125]: -1.42755177876459

In [126]: yo = sy.nsolve(del_y, -1.5) ❸
          yo ❸
Out[126]: -1.42755177876459

In [127]: f.subs({x : xo, y : yo}).evalf() ❹
Out[127]: -1.77572565314742
```

❶ 函数的符号版本。

❷ 得出并打印两个偏导数。

❸ 对根和结果最优值进行有根据的推测。

❹ 全局最小函数值。

若提供没有根据/随机的猜测，可能使算法陷入某个局部最小值，而非全局最小值：

```
In [128]: xo = sy.nsolve(del_x, 1.5) ❶
          xo
Out[128]: 1.74632928225285
```

```
In [129]: yo = sy.nsolve(del_y, 1.5) ❶
          yo
Out[129]: 1.74632928225285

In [130]: f.subs({x : xo, y : yo}).evalf() ❷
Out[130]: 2.27423381055640
```

❶ 根的有根据猜测。

❷ 局部最小函数值。

这从数值上说明了偏微分为 0 是必要条件，但并非充分条件。

> **符号计算**
> 用 Python 解决数学问题时，应该时刻考虑到 SymPy 和符号计算。特别是对于交互式的金融分析，这可能是比非符号方法更高效的方法。

11.5 结语

本章介绍了一些对金融学很重要的数学主题和工具。函数逼近在许多金融领域都很重要，例如基于曲线插值和回归的美式期权蒙特卡洛估值方法。凸优化技术在金融学中也经常用到，例如，用期权的市场报价或者隐含波动率来检验参数化期权定价模型。

数值积分是很多问题的中心，例如期权和衍生品的定价问题。得出一组随机过程的风险中立概率测度之后，期权的定价就可以归结于求取风险中立测度下的期权预期收益，并折现为当前日期的价值。第 12 章将介绍多种随机过程在风险中立测度下的模拟。

最后，本章介绍了使用 SymPy 进行符号计算的方法。对于一些数学运算（如积分、微分或者方程式的解），符号计算可以提供一种真正实用、高效的工具。

11.6 延伸阅读

本章使用的 Python 库的进一步信息可以参考如下在线资源。

- 本章使用的 NumPy 函数参见 SciPy 官方文档的 NumPy 介绍。
- 优化和求根的资料参见 SciPy 官方文档的 Optimization and Root Finding 介绍。

- 使用 `scipy.integrate` 求积分的方法参见 SciPy 官方文档的 Integration and ODEs 介绍。
- SymPy 网站提供丰富的示例和详细的文档。

下面是有关本章所介绍数学概念的一本很好的参考书：

- Brandimarte, Paolo (2006). *Numerical Methods in Finance and Economics*. Hoboken, NJ: John Wiley & Sons.

第 12 章

推断统计学

可预测性不是事物的走向，而是它们的可能趋势。

——拉海尔·法鲁克

当今，推断统计学是金融学中最重要的数学和数值学科之一。在现代金融学发端之时（20 世纪 70 年代和 80 年代），金融研究的主要目标是提出封闭式的解决方案，例如，在特定金融模型下的期权定价。近年来，需求有了根本的变化，不仅需要准确地估计对金融市场参与者很重要的单一金融工具的价值，还要一致性地估计整个衍生品谱系的价格。类似地，为了在整个金融机构中提出一致性的风险测度，例如风险价值和信用价值调整，需要将该机构及所有同行当成一个整体来考虑。这种令人头疼的任务只能通过灵活、高效的数值方法来处理。因此，推断统计学和作为特例的蒙特卡洛模拟被提升到了显要的地位。

本章从 Python 的角度介绍如下内容。

随机数

一切都从（伪）随机数开始，它是所有模拟工作的基础；尽管准随机数（例如，基于 Sobol 序列）在金融学中相当流行，但是伪随机数仍是基准。

模拟

在金融学中，两种模拟工作特别重要：随机变量和随机过程的模拟。

估值

估值有两种主要方式：欧式行权（特定日期）的衍生品估值和美式行权（在特定时期内）估值；另外还有用于百慕大行权（一组有限的特定日期）的估值工具。

风险测度

模拟本身需要借助风险测度的计算，例如风险价值、信用风险价值和信用价值调整。

12.1 随机数

在本章中，我们将用 `numpy.random` 子库提供的函数来生成随机数[1]：

```
In [1]: import math
        import numpy as np
        import numpy.random as npr ❶
        from pylab import plt, mpl

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

❶ 从 NumPy 导入随机数生成子库。

例如，`rand()`函数返回区间[0，1)的随机数，随机数的个数由参数指定。返回对象是一个 `ndarray` 对象。这些数值很容易转换以覆盖其余实际区间。例如，如果想生成区间[5，10)的随机数，可以像下例一样转换 `npr.rand()`的返回值——由于 NumPy 的广播特性，这也适合于多维数组：

```
In [3]: npr.seed(100) ❶
        np.set_printoptions(precision=4) ❶

In [4]: npr.rand(10) ❷
Out[4]: array([0.5434, 0.2784, 0.4245, 0.8448, 0.0047, 0.1216, 0.6707,
               0.8259, 0.1367, 0.5751])

In [5]: npr.rand(5, 5) ❸
Out[5]: array([[0.8913, 0.2092, 0.1853, 0.1084, 0.2197],
               [0.9786, 0.8117, 0.1719, 0.8162, 0.2741],
               [0.4317, 0.94  ,  0.8176, 0.3361, 0.1754],
               [0.3728, 0.0057, 0.2524, 0.7957, 0.0153],
               [0.5988, 0.6038, 0.1051, 0.3819, 0.0365]])

In [6]: a = 5. ❹
        b = 10. ❺
        npr.rand(10) * (b - a) + a ❻
Out[6]: array([9.4521, 9.9046, 5.2997, 9.4527, 7.8845, 8.7124, 8.1509,
               7.9092, 5.1022, 6.0501])
```

1　简单起见，我们将使用的所有伪随机数都称作随机数。——原注

```
In [7]: npr.rand(5, 5) * (b - a) + a ❼
Out[7]: array([[7.7234, 8.8456, 6.2535, 6.4295, 9.262 ],
               [9.875 , 9.4243, 6.7975, 7.9943, 6.774 ],
               [6.701 , 5.8904, 6.1885, 5.2243, 7.5272],
               [6.8813, 7.964 , 8.1497, 5.713 , 9.6692],
               [9.7319, 8.0115, 6.9388, 6.8159, 6.0217]])
```

❶ 固定种子值以便重现，固定打印输出的位数。

❷ 一维 ndarray 对象形式的均匀分布随机数。

❸ 二维 ndarray 对象形式的均匀分布随机数。

❹ 低限。

❺ 高限。

❻ 转换为另一个区间。

❼ 两个维度的相同转换。

表 12-1 列出了生成简单随机数的函数。

表 12-1　简单随机数生成函数

函　　数	参　　数	描　　述
rand	d0, d1, …, dn	指定组成的随机数
randn	d0, d1, …, dn	来自标准正态分布的一个（或者多个）样本
randint	low[, high, size]	从 low（含）到 high（不含）的随机整数
random_integers	low[, high, size]	low 和 high（含）之间的随机整数
random_sample	[size]	半开区间[0.0, 1.0)内的随机浮点数
random	[size]	半开区间[0.0, 1.0)内的随机浮点数
ranf	[size]	半开区间[0.0, 1.0)内的随机浮点数
sample	[size]	半开区间[0.0, 1.0)内的随机浮点数
choice	a[, size, replace, p]	给定一维数组中的随机样本
bytes	length	随机字节

表 12-1 中的函数所生成的随机数很容易可视化。图 12-1 以直观的方式展示了两个连续分布和两个离散分布的结果：

```
In [8]: sample_size = 500
        rn1 = npr.rand(sample_size, 3) ❶
        rn2 = npr.randint(0, 10, sample_size) ❷
        rn3 = npr.sample(size=sample_size) ❶
```

```
        a = [0, 25, 50, 75, 100] ❸
        rn4 = npr.choice(a, size=sample_size) ❸

In [9]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2,
                                                      figsize=(10, 8))
        ax1.hist(rn1, bins=25, stacked=True)
        ax1.set_title('rand')
        ax1.set_ylabel('frequency')
        ax2.hist(rn2, bins=25)
        ax2.set_title('randint')
        ax3.hist(rn3, bins=25)
        ax3.set_title('sample')
        ax3.set_ylabel('frequency')
        ax4.hist(rn4, bins=25)
        ax4.set_title('choice');
```

❶ 均匀分布随机数。

❷ 给定区间内的随机整数。

❸ 来自有限列表对象的随机采样值。

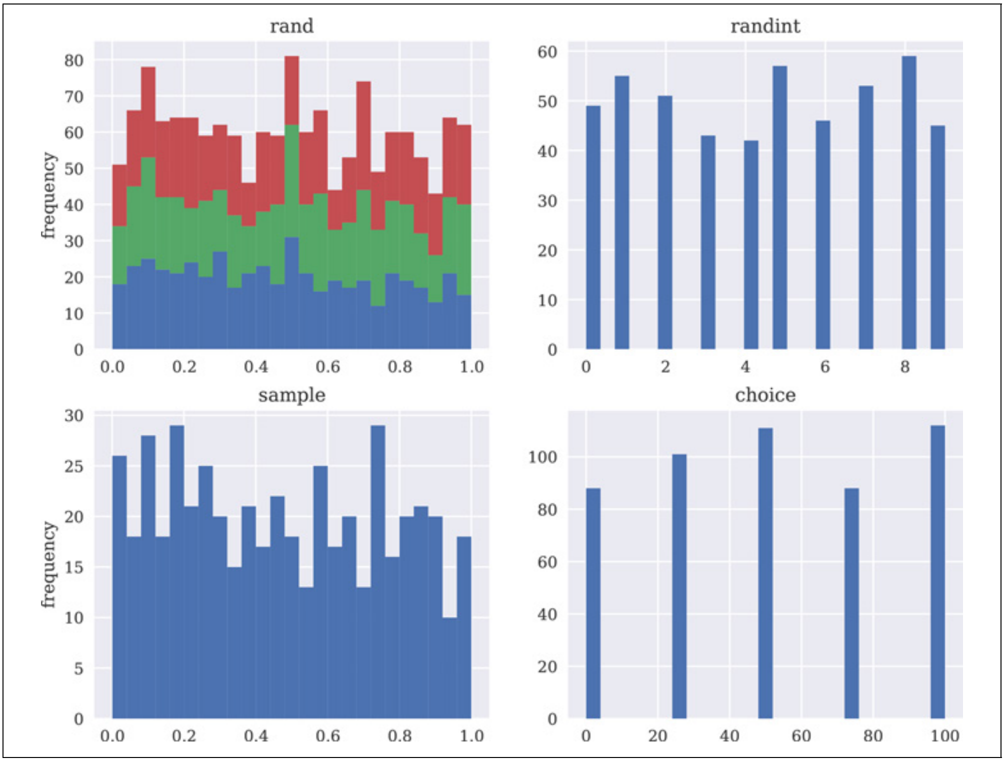


图 12-1　简单随机数的直方图

表 12-2 列出了根据不同分布生成随机数的函数。

表 12-2　根据不同分布生成随机数的函数

函　　数	参　　数	描　　述
beta	a, b[, size]	[0,1]区间上的 β 分布样本
binomial	n, p[, size]	二项分布样本
chisquare	df[, size]	卡方分布样本
dirichlet	alpha[, size]	狄利克雷分布样本
exponential	[scale, size]	指数分布样本
f	dfnum, dfden[, size]	F 分布样本
gamma	shape[, scale, size]	γ 分布样本
geometric	P[, size]	集合分布样本
gumbel	[loc, scale, size]	冈贝尔分布样本
hypergeometric	ngood, nbad, nsample[,size]	超几何分布样本
laplace	[loc, scale, size]	拉普拉斯分布或者双指数分布样本
logistic	[loc, scale, size]	逻辑分布样本
Lognormal	[mean, sigma, size]	对数正态分布样本
logseries	P[, size]	对数序列分布样本
multinomial	N, pvals[, size]	多项分布样本
multivariate_normal	mean, cov[, size]	多变量正态分布样本
negative_binomial	n, p[, size]	负二项式分布样本
noncentral_chisquare	df, nonc[, size]	非中心卡方分布样本
noncentral_f	dfnum, dfden, nonc[, size]	非中心 F 分布样本
normal	[loc, scale, size]	正态（高斯）分布样本
pareto	A[, size]	特定组成的帕累托 II 或者洛马克思分布样本
poisson	[lam, size]	泊松分布样本
power	A[, size]	[0,1]区间内指数为正（a−1）的幂次分布样本
rayleigh	[scale, size]	瑞利分布样本
standard_cauchy	[size]	标准柯西分布（模式 0）样本

续表

函　　数	参　　数	描　　述
standard_exponential	[size]	标准指数分布样本
standard_gamma	shape[, size]	标准 γ 分布样本
standard_normal	[size]	标准正态分布（均值为 0，标准差为 1）样本
standard_t	df[, size]	学生的 t 分布样本（自由度为 df）
triangular	left, mode, right[, size]	三角分布样本
uniform	[low, high, size]	均匀分布样本
vonmises	mu, kappa[, size]	冯・米塞斯分布样本
wald	mean, scale[, size]	瓦尔德（逆高斯）分布样本
weibull	a[, size]	威布尔分布样本
zipf	a[, size]	齐夫分布样本

虽然在金融学中使用（标准）正态分布遭到了许多批评，但它们仍然是不可或缺的工具，在分析和数值应用中仍然是最广泛使用的分布类型之一。原因之一是许多金融模型直接依赖于正态分布或者对数正态分布。另一个原因是许多不直接依赖（对数）正态假设的金融模型可以离散化，从而使用正态分布进行近似模拟。

作为例子，图 12-2 可视化了如下分布的随机数：

- 均值为 0，标准差为 1 的标准正态分布；
- 均值为 100，标准差为 20 的正态分布；
- 自由度为 0.5 的卡方分布；
- λ 值为 1 的泊松分布。

图 12-2 展示了 3 个连续分布和 1 个离散分布（泊松分布）的结果。泊松分布用于模拟（罕见）外部事件的发生，例如某种金融工具价格暴涨或者外部冲击。下面是生成图形的代码：

```
In [10]: sample_size = 500
         rn1 = npr.standard_normal(sample_size) ❶
         rn2 = npr.normal(100, 20, sample_size) ❷
         rn3 = npr.chisquare(df=0.5, size=sample_size) ❸
         rn4 = npr.poisson(lam=1.0, size=sample_size) ❹

In [11]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2,
                                                       figsize=(10, 8))
         ax1.hist(rn1, bins=25)
```

```
        ax1.set_title('standard normal')
        ax1.set_ylabel('frequency')
        ax2.hist(rn2, bins=25)
        ax2.set_title('normal(100, 20)')
        ax3.hist(rn3, bins=25)
        ax3.set_title('chi square')
        ax3.set_ylabel('frequency')
        ax4.hist(rn4, bins=25)
        ax4.set_title('Poisson');
```

❶ 标准正态分布随机数。

❷ 正态分布随机数。

❸ 卡方分布随机数。

❹ 泊松分布随机数。

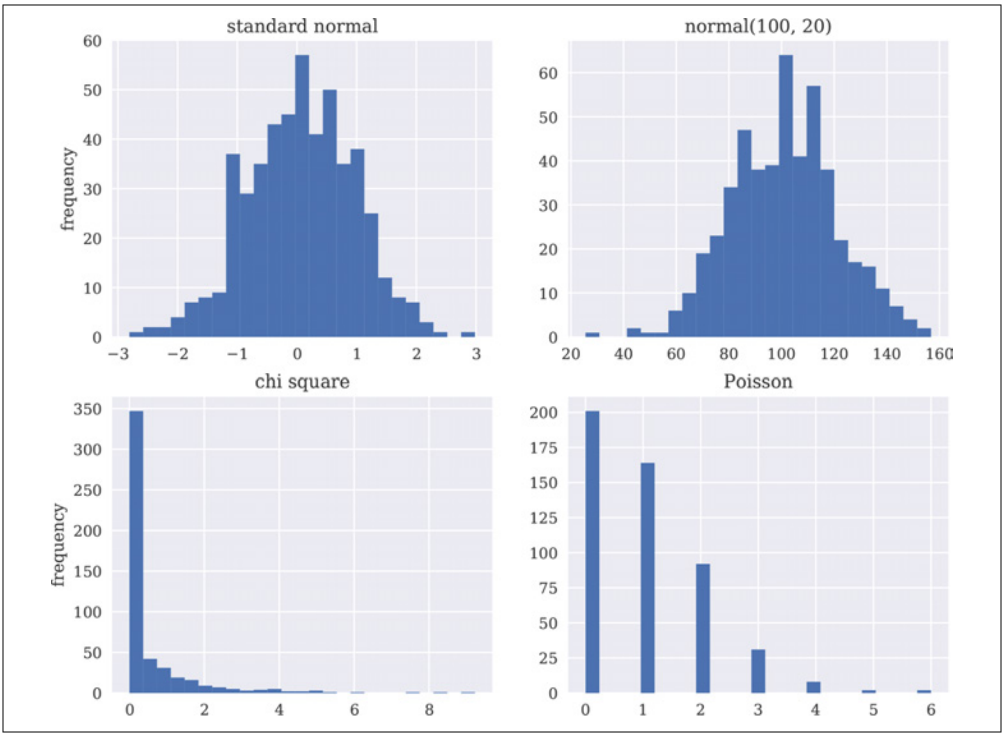


图 12-2　不同分布随机样本的直方图

NumPy 与随机数

本节说明 NumPy 是一个强大（甚至不可或缺）的 Python 伪随机数生成工具。创建包含这些数值的 ndarray 对象不仅方便，而且高效。

12.2　模拟

蒙特卡洛模拟（MCS）是金融学中最重要的数值技术之一（在重要性和使用广泛程度上也许没有"之一"）。这主要是因为它是最灵活的数学表达式（如积分）求值方法，特别适合于金融衍生品的估值。但是，这种灵活性的代价是相对高的计算负担，估算一个值就可能需要数十万次甚至数百万次的复杂计算。

12.2.1　随机变量

举个例子，我们考虑期权定价所用的 Black-Scholes-Merton 设置（参见第 3 章）。在这种设置中，在今日股票指数水平 S_0 给定的情况下，未来某个日期 T 的股票指数水平 S_T 可根据公式 12-1 得出。

公式 12-1　以 Black-Scholes-Merton 设置模拟未来指数水平

$$S_T = S_0 \exp\left(\left(r - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}z\right)$$

变量和参数的含义如下。

S_T

　　T 日的指数水平。

r

　　恒定无风险短期利率。

σ

　　S 的恒定波动率（=收益的标准差）。

z

　　标准正态分布随机变量。

这种简单金融模型的参数化和模拟代码如下，输出如图 12-3 所示：

```
In [12]: S0 = 100 ❶
         r = 0.05 ❷
         sigma = 0.25 ❸
         T = 2.0 ❹
         I = 10000 ❺
         ST1 = S0 * np.exp((r - 0.5 * sigma ** 2) * T +
                 sigma * math.sqrt(T) * npr.standard_normal(I)) ❻
```

```
In [13]: plt.figure(figsize=(10, 6))
         plt.hist(ST1, bins=50)
         plt.xlabel('index level')
         plt.ylabel('frequency');
```

❶ 初始指数水平。

❷ 恒定无风险短期利率。

❸ 恒定波动率因子。

❹ 以年表示的时限。

❺ 模拟数量。

❻ 模拟本身通过一个向量化表达式进行，离散格式使用 `npr.standard_normal()` 函数。

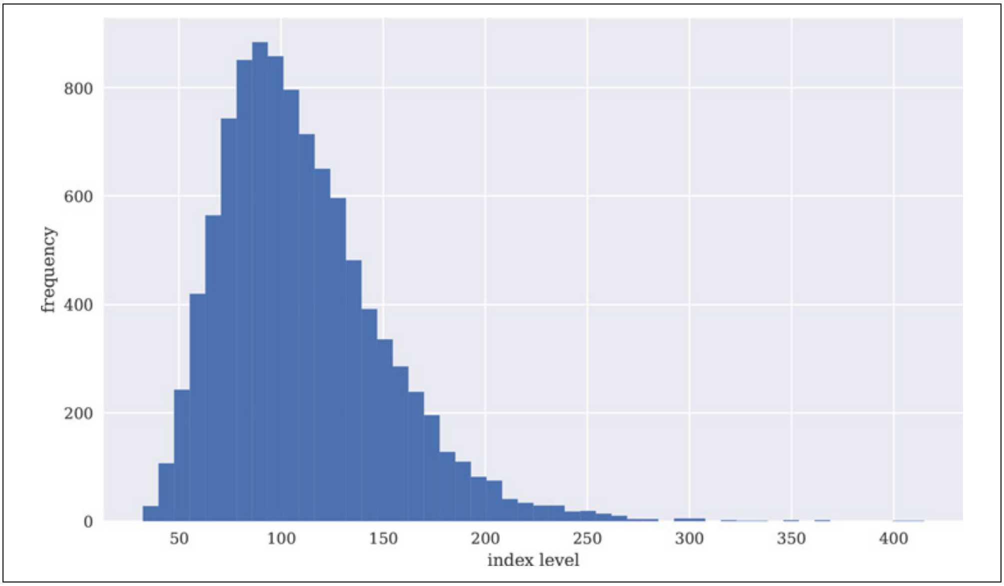


图 12-3 （通过 `npr.standard_normal()`）统计模拟的几何布朗运动

从图 12-3 中可以看出，公式 12-1 中定义的随机变量呈对数正态分布。因此，我们还可以尝试使用 `npr.lognormal()` 函数直接得出随机变量值。在这种情况下，我们必须向函数提供均值和标准差：

```
In [14]: ST2 = S0 * npr.lognormal((r - 0.5 * sigma ** 2) * T,
                                  sigma * math.sqrt(T), size=I) ❶

In [15]: plt.figure(figsize=(10, 6))
         plt.hist(ST2, bins=50)
```

```
        plt.xlabel('index level')
        plt.ylabel('frequency');
```

❶ 模拟通过一个向量化表达式进行，离散格式使用 `npr.lognormal()`函数。

结果如图 12-4 所示。

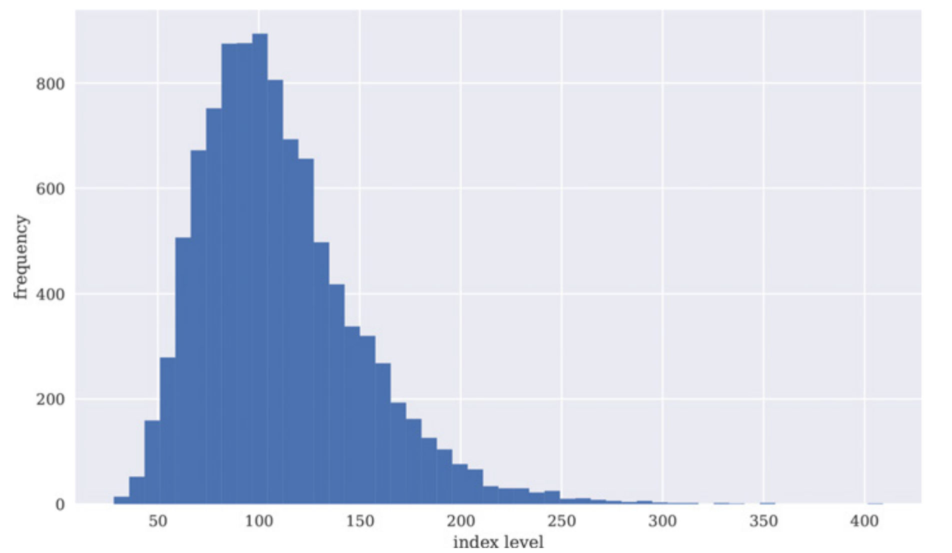


图 12-4 （通过 **npr.lognormal()**）模拟几何布朗运动

比较图 12-3 和图 12-4，可看出它们确实很相似。但是我们还将通过比较结果分布的统计矩来更严格地检验。

事实证明，`scipy.stats` 子库和下面定义的助手函数 `print_statistics()`，能够有效地比较模拟结果的分布特性：

```
In [16]: import scipy.stats as scs

In [17]: def print_statistics(a1, a2):
             ''' Prints selected statistics.

             Parameters
             ==========
             a1, a2: ndarray objects
                 results objects from simulation
             '''
             sta1 = scs.describe(a1) ❶
             sta2 = scs.describe(a2) ❶
             print('%14s %14s %14s' %
                 ('statistic', 'data set 1', 'data set 2'))
```

```
    print(45 * "-")
    print('%14s %14.3f %14.3f' % ('size', sta1[0], sta2[0]))
    print('%14s %14.3f %14.3f' % ('min', sta1[1][0], sta2[1][0]))
    print('%14s %14.3f %14.3f' % ('max', sta1[1][1], sta2[1][1]))
    print('%14s %14.3f %14.3f' % ('mean', sta1[2], sta2[2]))
    print('%14s %14.3f %14.3f' % ('std', np.sqrt(sta1[3]),
                                  np.sqrt(sta2[3])))
    print('%14s %14.3f %14.3f' % ('skew', sta1[4], sta2[4]))
    print('%14s %14.3f %14.3f' % ('kurtosis', sta1[5], sta2[5]))

In [18]: print_statistics(ST1, ST2)
              statistic     data set 1     data set 2
         ---------------------------------------------
                   size      10000.000      10000.000
                    min         32.327         28.230
                    max        414.825        409.110
                   mean        110.730        110.431
                    std         40.300         39.878
                   skew          1.122          1.115
               kurtosis          2.438          2.217
```

❶ `scs.describe()`函数给出数据集的重要统计量。

显然，两个模拟结果的特性很类似，差异主要是由于模拟中所谓的采样误差。在离散地模拟连续随机过程时会引入离散化误差，但是由于模拟方法的特性，这种误差在此不起任何作用。

12.2.2 随机过程

粗略地讲，随机过程是一个随机变量序列。从这个意义上看，我们应该预期：在模拟一个过程时，随机过程与一个随机变量序列的重复模拟应该有某种类似之处。这个结论大体上是正确的，但是随机数的选取一般不是独立的，而是依赖于前几次选取的结果。不过，金融学中使用的随机过程通常表现出马尔科夫特性——主要的含义是：明天的过程值只依赖于今天的过程状态，而不依赖其他任何“历史”状态，甚至不依赖整个路径历史。这种过程也被称作“无记忆过程”。

1. 几何布朗运动

现在考虑 Black-Scholes-Merton 模型的动态形式，这种形式由公式 12-2 中的随机微分方程（SDE）描述。式中的 Z_t 是标准布朗运动，SDE 被称作几何布朗运动。S_t 的值呈对数正态分布，（边际）收益 dS_t/S_t 呈正态分布。

公式 12-2　Black-Scholes-Merton 设置中的随机微分方程

$$dS_t = rS_t dt + \sigma S_t dZ_t$$

公式 12-2 中的 SDE 可以由一个欧拉格式精确地离散化，公式 12-3 中介绍了一个这样的格式，其中 Δt 是固定的离散化间隔，z_t 是标准正态分布随机变量。

公式 12-3　在 Black-Scholes-Merton 设置中动态模拟指数水平

$$S_t = S_{t-\Delta t} \exp\left(\left(r - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\sqrt{\Delta t} z_t\right)$$

和以前一样，转换为 Python 和 NumPy 的代码很简单。得出的指数水平终值仍然呈对数正态分布，如图 12-5 所示。

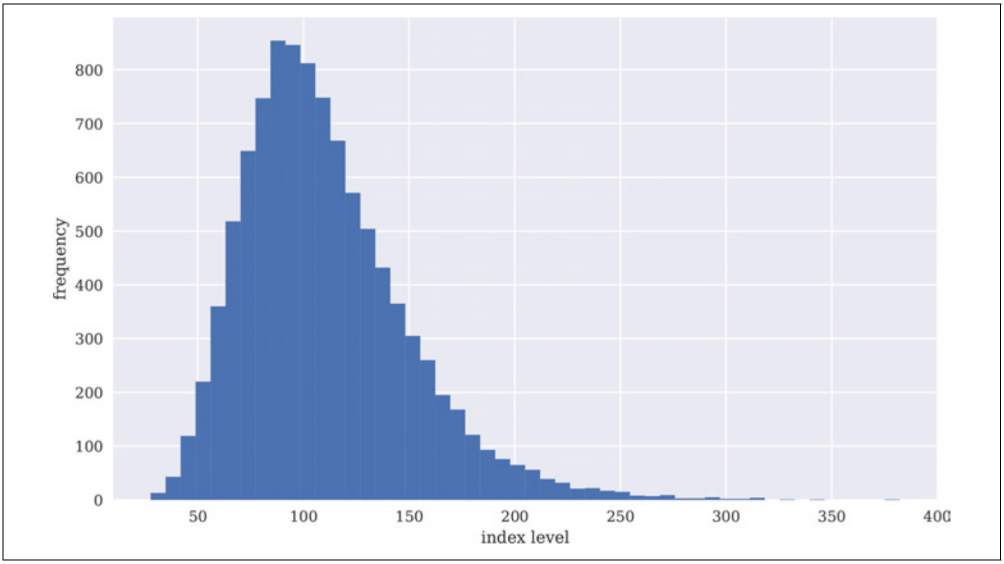


图 12-5　动态模拟的到期日集合布朗运动

前 4 个统计矩和静态模拟方法得出的结果相当接近：

```
In [19]: I = 10000 ❶
         M = 50 ❷
         dt = T / M ❸
         S = np.zeros((M + 1, I)) ❹
         S[0] = S0 ❺
         for t in range(1, M + 1):
             S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt +
                     sigma * math.sqrt(dt) * npr.standard_normal(I)) ❻

In [20]: plt.figure(figsize=(10, 6))
         plt.hist(S[-1], bins=50)
         plt.xlabel('index level')
         plt.ylabel('frequency');
```

❶ 模拟路径数量。

❷ 离散化所用的时间间隔数量。

❸ 以年表示的时间间隔长度。

❹ 保存指数水平的二维 ndarray 对象。

❺ 时间 $t = 0$（起始点）的初始值。

❻ 模拟通过半向量化表达式进行；循环从 $t = 1$ 的时点开始，$t = T$ 时结束。

下面是动态模拟和静态模拟所得统计量的对比。图 12-6 展示了前 10 条模拟路径：

```
In [21]: print_statistics(S[-1], ST2)
              statistic     data set 1    data set 2
         ---------------------------------------------
                   size      10000.000     10000.000
                    min         27.746        28.230
                    max        382.096       409.110
                   mean        110.423       110.431
                    std         39.179        39.878
                   skew          1.069         1.115
               kurtosis          2.028         2.217

In [22]: plt.figure(figsize=(10, 6))
         plt.plot(S[:, :10], lw=1.5)
         plt.xlabel('time')
         plt.ylabel('index level');
```



图 12-6　动态模拟几何布朗运动路径

使用动态模拟方法不仅可以像图 12-6 那样可视化路径，还可以估算美式/百慕大期权或者收益与路径相关的期权价值。可以这么说，你所得到的是全动态图像。

2. 平方根扩散

另一类重要的金融过程是均值回归过程，用于建立短期利率或者波动性过程的模型。流行和广泛使用的模型之一是平方根扩散，由 Cox、Ingersoll 和 Ross（1985）提出。公式 12-4 提供了对应的 SDE。

公式 12-4　平方根扩散的随机微分方程

$$\mathrm{d}x_t = k(\theta - x_t)\mathrm{d}t + \sigma\sqrt{x_t}\mathrm{d}Z_t$$

式中的变量和参数的含义如下。

x_t

　　日期 t 的过程水平。

k

　　均值回归因子。

θ

　　长期过程均值。

σ

　　恒定波动率参数。

Z_t

　　标准布朗运动。

众所周知，x_t 的值呈卡方分布。但是，如前所述，许多金融模型可以使用正态分布进行离散化和近似计算（即所谓的欧拉离散化格式）。虽然欧拉格式对几何布朗运动很精确，但是对于大部分其他随机过程则会产生偏差。即使有精确的格式（很快就会介绍一个用于平方根扩散的格式）但因为数值化和/或计算，使用欧拉格式仍是最合适的。定义 $s = t - \Delta t$ 和 $x^+ \equiv \max(x,0)$，公式 12-5 提出了一种欧拉格式。这种特殊格式在文献中通常称作完全截断。

公式 12-5　平方根扩散的欧拉离散化

$$\begin{aligned}\tilde{x}_t &= \tilde{x}_s + k(\theta - \tilde{x}_s^+)\Delta t + \sigma\sqrt{\tilde{x}_s^+}\sqrt{\Delta t}z_t \\ x_t &= \tilde{x}_t^+\end{aligned}$$

平方根扩散的特性为方便和现实——x_t 的值严格为正。用欧拉格式离散化时，负值无法

排除。这就是人们处理的总是原始模拟过程正数版本的原因。因此在模拟代码中，需要两个 ndarray 对象，而不是一个。图 12-7 用直方图直观地显示了模拟结果：

```
In [23]: x0 = 0.05 ❶
         kappa = 3.0 ❷
         theta = 0.02 ❸
         sigma = 0.1 ❹
         I = 10000
         M = 50
         dt = T / M

In [24]: def srd_euler():
             xh = np.zeros((M + 1, I))
             x = np.zeros_like(xh)
             xh[0] = x0
             x[0] = x0
             for t in range(1, M + 1):
             xh[t] = (xh[t - 1] +
                      kappa * (theta - np.maximum(xh[t - 1], 0)) * dt +
                      sigma * np.sqrt(np.maximum(xh[t - 1], 0)) *
                      math.sqrt(dt) * npr.standard_normal(I)) ❺
         x = np.maximum(xh, 0)
         return x
     x1 = srd_euler()

In [25]: plt.figure(figsize=(10, 6))
         plt.hist(x1[-1], bins=50)
         plt.xlabel('value')
         plt.ylabel('frequency');
```

❶ 初始值（例如短期利率）。

❷ 均值回复因数。

❸ 长期均值。

❹ 波动率因数。

❺ 基于欧拉格式的模拟。

图 12-8 展示了前 10 条模拟路径，说明得出的平均偏离值为负值（因为 $x_0 > \theta$）并收敛于 $\theta = 0.02$：

```
In [26]: plt.figure(figsize=(10, 6))
         plt.plot(x1[:, :10], lw=1.5)
         plt.xlabel('time')
         plt.ylabel('index level');
```

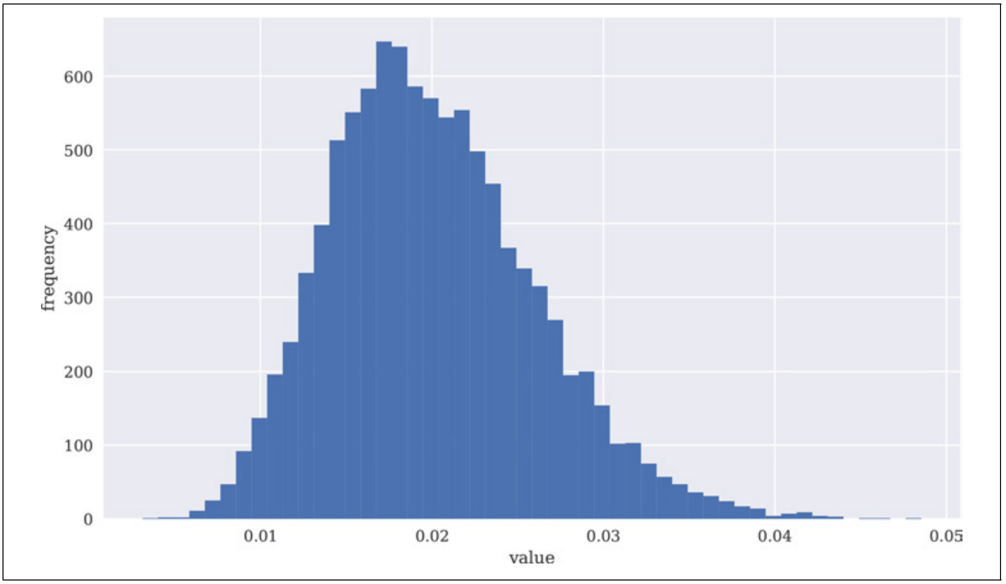


图 12-7　到期日的动态模拟平方根扩散（欧拉格式）

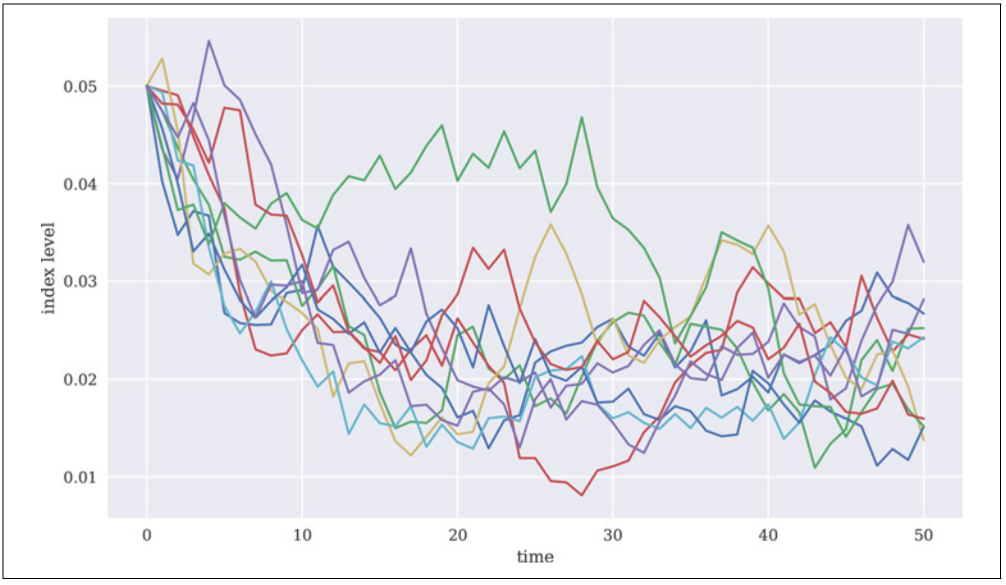


图 12-8　动态模拟平方根扩散路径（欧拉格式）

公式 12-6 提出了基于自由度 $\mathrm{d}f = \frac{4\theta k}{\sigma^2}$、非中心参数 $nc = \frac{4k\mathrm{e}^{-k\Delta t}}{\sigma^2(1-\mathrm{e}^{-k\Delta t})}x_s$ 的卡方分布 $\chi_d'^2$ 平方根扩散的精确离散化格式。

公式 12-6　平方根扩散的精确离散化

$$x_t = \frac{\sigma^2(1-e^{-k\Delta t})}{4k} x_d'^2 \left(\frac{4ke^{-k\Delta t}}{\sigma^2(1-e^{-k\Delta t})} x_s \right)$$

上述离散化格式的 Python 实现较为复杂，但是仍然很简洁。图 12-9 以直方图形式展示了这个精确格式的模拟输出：

```
In [27]: def srd_exact():
             x = np.zeros((M + 1, I))
             x[0] = x0
             for t in range(1, M + 1):
                 df = 4 * theta * kappa / sigma ** 2 ❶
                 c = (sigma ** 2 * (1 - np.exp(-kappa * dt))) / (4 *
kappa) ❶
                 nc = np.exp(-kappa * dt) / c * x[t - 1] ❶
                 x[t] = c * npr.noncentral_chisquare(df, nc, size=I) ❶
             return x
         x2 = srd_exact()

In [28]: plt.figure(figsize=(10, 6))
         plt.hist(x2[-1], bins=50)
         plt.xlabel('value')
         plt.ylabel('frequency');
```

❶ 使用 `npr.concentral_chisquare()` 函数的离散化格式。

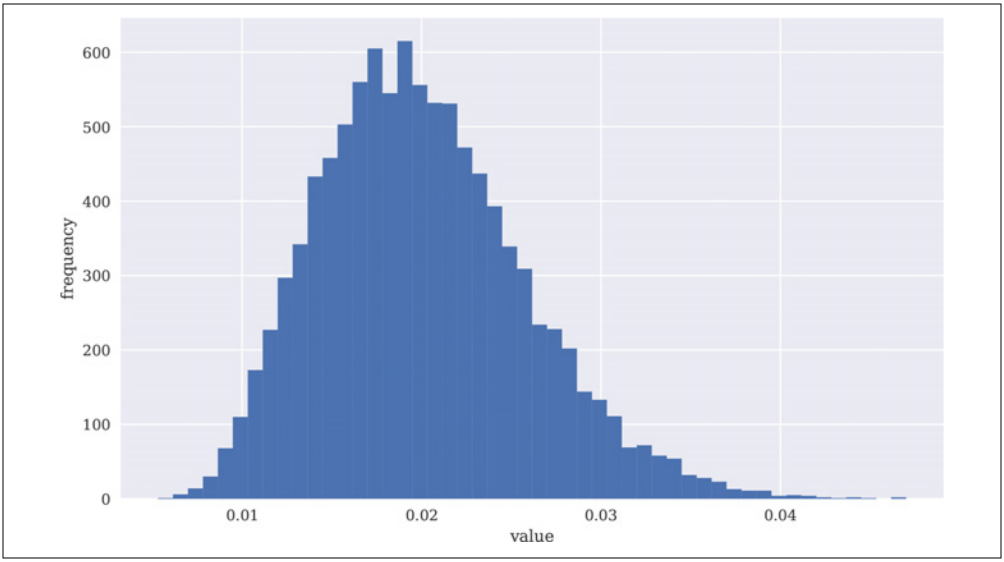


图 12-9　到期日的动态模拟平方根扩散（精确格式）

图 12-10 和以前一样提供了 10 条模拟路径，同样呈现负的平均漂移，收敛于 θ：

```
In [29]: plt.figure(figsize=(10, 6))
         plt.plot(x2[:, :10], lw=1.5)
         plt.xlabel('time')
         plt.ylabel('index level');
```



图 12-10　动态模拟平方根扩散路径（精确格式）

比较不同方法中的主要统计量可以看出，偏置欧拉格式确实很好地表现出了理想的统计属性：

```
In [30]: print_statistics(x1[-1], x2[-1])
              statistic     data set 1      data set 2
         ---------------------------------------------
                   size      10000.000       10000.000
                    min          0.003           0.005
                    max          0.049           0.047
                   mean          0.020           0.020
                    std          0.006           0.006
                   skew          0.529           0.532
               kurtosis          0.289           0.273

In [31]: I = 250000
         %time x1 = srd_euler()
         CPU times: user 1.62 s, sys: 184 ms, total: 1.81 s
         Wall time: 1.08 s
```

```
In [32]: %time x2 = srd_exact()
         CPU times: user 3.29 s, sys: 39.8 ms, total: 3.33 s
         Wall time: 1.98 s

In [33]: print_statistics(x1[-1], x2[-1])
         x1 = 0.0; x2 = 0.0
              statistic     data set 1      data set 2
         ---------------------------------------------
                   size     250000.000      250000.000
                    min          0.002           0.003
                    max          0.071           0.055
                   mean          0.020           0.020
                    std          0.006           0.006
                   skew          0.563           0.579
               kurtosis          0.492           0.520
```

然而，在执行速度方面它们有较大的差异，这是因为从非中心卡方分布中采样的计算要求高于标准正态分布的采样。精确格式大约需要花费 2 倍的时间，而结果实际上和欧拉格式相同。

3. 随机波动率

Black-Scholes-Merton 模型中重要的简化假设之一是恒定波动率。但是，波动率一般来说既不是恒定的、也不具有确定性，而是随机的。因此，20 世纪 90 年代初金融模型的重大进步之一是所谓随机波动率模型的推出。这一类别中最为流行的模型之一是 Heston（1993）模型，如公式 12-7 所示。

公式 12-7　Heston 随机波动率模型的随机微分方程

$$
\begin{aligned}
\mathrm{d}S_t &= rS_t\mathrm{d}t + \sqrt{v_t}S_t\mathrm{d}Z_t^1 \\
\mathrm{d}v_t &= k_v(\theta_v - v_t)\mathrm{d}t + \sigma_v\sqrt{v_t}\mathrm{d}Z_t^2 \\
\mathrm{d}Z_t^1\mathrm{d}Z_t^2 &= \rho
\end{aligned}
$$

单一变量和参数的均值现在很容易从几何布朗运动和平方根扩散的讨论中得出，参数 ρ 代表两个标准布朗运动 Z_t^1 、Z_t^2 之间的瞬时相关性。这使我们可以解释一个典型事实——杠杆效应，该效应本质上指的是波动性在困难时期（衰退市场）中上升，而在牛市（上升市场）时下降。

考虑上述模型的参数化。为了说明两个随机过程之间的相关性，我们需要确定相关矩阵的柯列斯基分解：

```
In [34]: S0 = 100.
         r = 0.05
         v0 = 0.1 ❶
```

```
         kappa = 3.0
         theta = 0.25
         sigma = 0.1
         rho = 0.6 ❷
         T = 1.0

In [35]: corr_mat = np.zeros((2, 2))
         corr_mat[0, :] = [1.0, rho]
         corr_mat[1, :] = [rho, 1.0]
         cho_mat = np.linalg.cholesky(corr_mat) ❸

In [36]: cho_mat ❸
Out[36]: array([[1. , 0. ],
                [0.6, 0.8]])
```

❶ 初始（瞬时）波动率值。

❷ 两个布朗运动之间的固定相关系数。

❸ 柯列斯基分解与结果矩阵。

在开始模拟随机过程之前，我们为两个过程生成整组随机数，指数过程使用第 0 组，波动性过程使用第 1 组。对于以平方根扩散过程类型建模的波动性过程，我们使用欧拉格式，通过柯列斯基矩阵考虑相关性参数：

```
In [37]: M = 50
         I = 10000
         dt = T / M

In [38]: ran_num = npr.standard_normal((2, M + 1, I)) ❶

In [39]: v = np.zeros_like(ran_num[0])
         vh = np.zeros_like(v)

In [40]: v[0] = v0
         vh[0] = v0

In [41]: for t in range(1, M + 1):
             ran = np.dot(cho_mat, ran_num[:, t, :]) ❷
             vh[t] = (vh[t - 1] +
                      kappa * (theta - np.maximum(vh[t - 1], 0)) * dt +
                      sigma * np.sqrt(np.maximum(vh[t - 1], 0)) *
                      math.sqrt(dt) * ran[1]) ❸

In [42]: v = np.maximum(vh, 0)
```

❶ 生成三维随机数数据集。

❷ 选择相关的随机数子集，并通过柯列斯基矩阵转换。

❸ 根据欧拉格式模拟路径。

对于指数水平过程，我们也考虑相关性，并使用几何布朗运动的精确欧拉格式。图 12-11 用直方图展示了到期日指数水平过程和波动性过程的模拟结果。

```
In [43]: S = np.zeros_like(ran_num[0])
         S[0] = S0
         for t in range(1, M + 1):
             ran = np.dot(cho_mat, ran_num[:, t, :])
             S[t] = S[t - 1] * np.exp((r - 0.5 * v[t]) * dt +
                                 np.sqrt(v[t]) * ran[0] * np.sqrt(dt))

In [44]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 6))
         ax1.hist(S[-1], bins=50)
         ax1.set_xlabel('index level')
         ax1.set_ylabel('frequency')
         ax2.hist(v[-1], bins=50)
         ax2.set_xlabel('volatility');
```

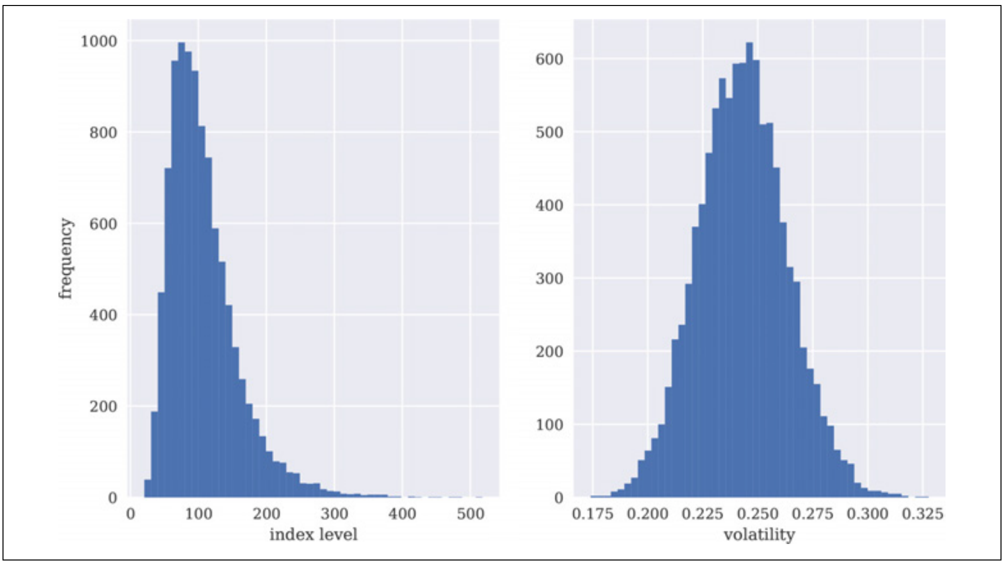


图 12-11　到期日的动态模拟随机波动率模型

这说明了在平方根扩散中使用欧拉格式的另一项优势：相关性很容易一致性地处理，这因为我们只提取标准正态随机数。没有一种混合方法（如对指数使用欧拉格式，对波动性过程使用基于非中心卡方分布的精确方法）能够简单地实现相同的效果。

对每个过程的前 10 条模拟路径（见图 12-12）的检查表明，波动性过程的平均漂移值为正数，和预期的一样收敛于 $\theta = 0.25$：

```
In [45]: print_statistics(S[-1], v[-1])
              statistic     data set 1     data set 2
         ---------------------------------------------
                   size      10000.000      10000.000
                    min         20.556          0.174
                    max        517.798          0.328
                   mean        107.843          0.243
                    std         51.341          0.020
                   skew          1.577          0.124
               kurtosis          4.306          0.048

In [46]: fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True,
                                        figsize=(10, 6))
         ax1.plot(S[:, :10], lw=1.5)
         ax1.set_ylabel('index level')
         ax2.plot(v[:, :10], lw=1.5)
         ax2.set_xlabel('time')
         ax2.set_ylabel('volatility');
```

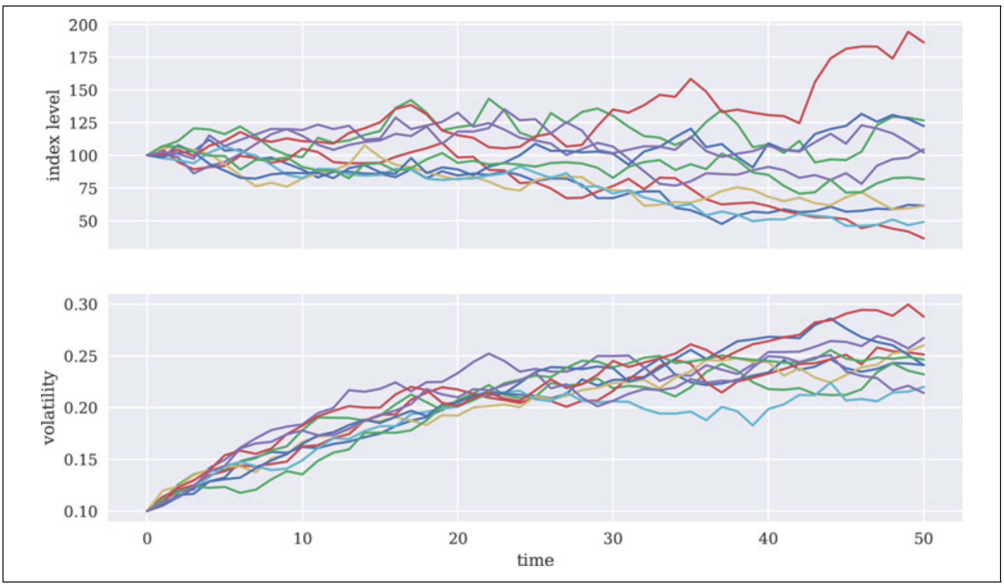


图 12-12　模拟随机波动率模型路径

最后，我们简短地看看两个数据集上最后一个时间点的统计量，该时间点显示指数水平过程有一个相当高的最大值。实际上，在其他条件不变的情况下，这个最大值远大

于固定波动率下几何布朗运动所能达到的最大值。

4. 跳跃扩散

随机波动率和杠杆效应是在许多市场上都能发现的典型（经验主义）事实。另一种典型的经验主义事实是资产价格和波动率的跳跃。1976 年，Merton 发布了他的跳跃扩散模型，模型的一个部分以对数正态分布生成跳跃，改进了 Black-Scholes-Merton 设置。风险中立 SDE 如公式 12-8 所示。

公式 12-8　Merton 跳跃扩散模型的随机微分方程

$$dS_t = (r - r_j)S_t dt + \sigma S_t dZ_t + J_t S_t dN_t$$

完整性起见，下面列出变量和参数的含义。

S_t

　　t 日的指数水平。

t

　　恒定无风险短期利率。

$$r_J \equiv \lambda \cdot (e^{u_J + \delta^2/2} - 1)$$

　　维持风险中立性的跳跃漂移校正。

σ

　　S 的恒定波动率。

Z_t

　　标准布朗运动。

J_t

　　t 日呈……分布的跳跃。

- $\log(1 + J_t) \approx N\left(\log(1 + \mu_J) - \frac{\delta^2}{2}, \delta^2\right)$。
- N 是标准正态随机变量的累积分布函数。

N_t

　　密度为 λ 的泊松分布。

公式 12-9 介绍了一种用于跳跃扩散的欧拉离散化公式，其中 z_t^n 呈标准正态分布，y_t 呈密度为 λ 的泊松分布。

公式 12-9　Merton 跳跃扩散模型的欧拉离散化

$$S_t = S_{t-\Delta t}\left(e^{(r-r_J-\sigma^2/2)\Delta t+\sigma\sqrt{\Delta t}z_t^1}+(e^{\mu_J+\delta z_t^2}-1)y_t\right)$$

有了上述的离散化格式，考虑下面的数值参数：

```
In [47]: S0 = 100.
         r = 0.05
         sigma = 0.2
         lamb = 0.75 ❶
         mu = -0.6 ❷
         delta = 0.25 ❸
         rj = lamb * (math.exp(mu + 0.5 * delta ** 2) - 1) ❹

In [48]: T = 1.0
         M = 50
         I = 10000
         dt = T / M
```

❶ 跳跃密度。

❷ 跳跃幅度均值。

❸ 跳跃波动率。

❹ 漂移校正。

这一次，我们需要 3 组随机数。注意，图 12-13 中的第二个波峰（双峰频率分布）是由于跳跃出现的：

```
In [49]: S = np.zeros((M + 1, I))
         S[0] = S0
         sn1 = npr.standard_normal((M + 1, I)) ❶
         sn2 = npr.standard_normal((M + 1, I)) ❶
         poi = npr.poisson(lamb * dt, (M + 1, I)) ❷
         for t in range(1, M + 1, 1):
             S[t] = S[t - 1] * (np.exp((r - rj - 0.5 * sigma ** 2) * dt +
                                sigma * math.sqrt(dt) * sn1[t]) +
                                (np.exp(mu + delta * sn2[t]) - 1) *
                                poi[t]) ❸
             S[t] = np.maximum(S[t], 0)

In [50]: plt.figure(figsize=(10, 6))
         plt.hist(S[-1], bins=50)
         plt.xlabel('value')
         plt.ylabel('frequency');
```

❶ 标准正态分布随机数。

❷ 泊松分布随机数。

❸ 基于精确欧拉格式的模拟。

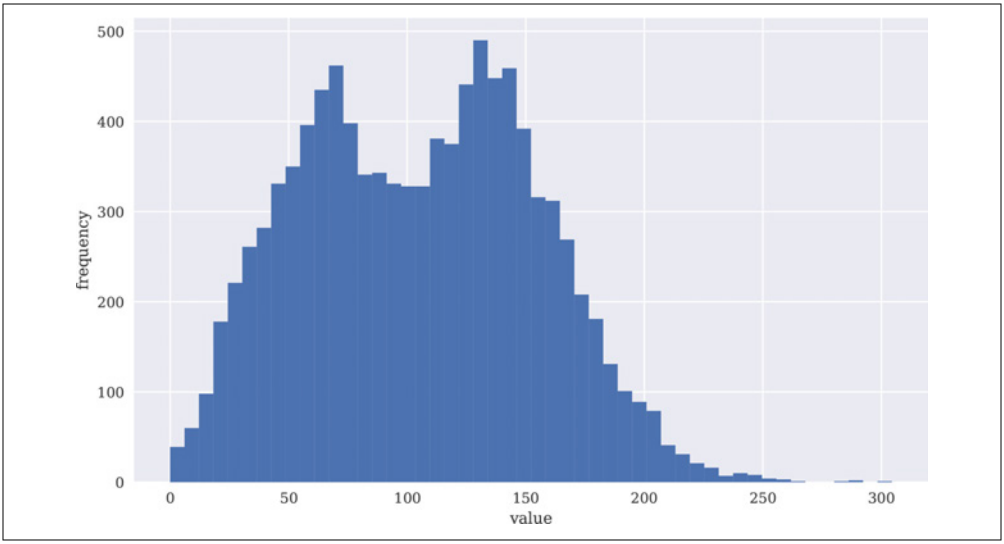


图 12-13　到期日的动态模拟跳跃扩散过程

在前 10 条模拟指数水平路径中也可以看到负数的跳跃均值，如图 12-14 所示：

```
In [51]: plt.figure(figsize=(10, 6))
         plt.plot(S[:, :10], lw=1.5)
         plt.xlabel('time')
         plt.ylabel('index level');
```

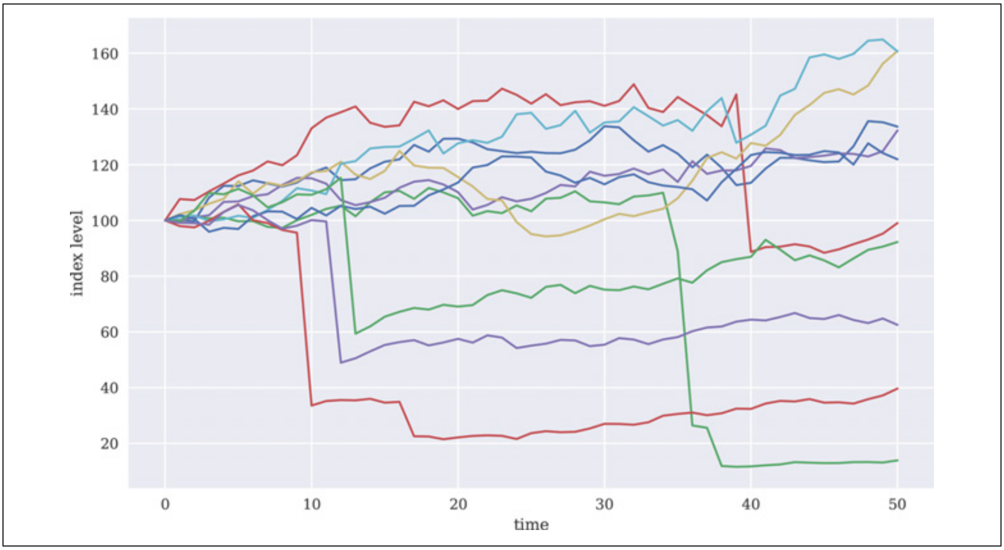


图 12-14　动态模拟跳跃扩散过程路径

12.2.3　方差缩减

目前为止，我们得到的结果表现出来的统计数值和预期/理想值并不足够接近，这不仅是因为使用的 Python 函数生成的是伪随机数，还因为提取的样本大小各不相同。例如，你可能预期一组标准正态分布随机数的均值为 0，标准差为 1。我们来看看不同组随机数的统计量。为了让比较更真实，我们修改随机数生成器的种子值：

```
In [52]: print('%15s %15s' % ('Mean', 'Std. Deviation'))
         print(31 * '-')
         for i in range(1, 31, 2):
             npr.seed(100)
             sn = npr.standard_normal(i ** 2 * 10000)
             print('%15.12f %15.12f' % (sn.mean(), sn.std()))
                    Mean  Std. Deviation
         -------------------------------
          0.001150944833  1.006296354600
          0.002841204001  0.995987967146
          0.001998082016  0.997701714233
          0.001322322067  0.997771186968
          0.000592711311  0.998388962646
         -0.000339730751  0.998399891450
         -0.000228109010  0.998657429396
          0.000295768719  0.998877333340
          0.000257107789  0.999284894532
         -0.000357870642  0.999456401088
         -0.000528443742  0.999617831131
         -0.000300171536  0.999445228838
         -0.000162924037  0.999516059328
          0.000135778889  0.999611052522
          0.000182006048  0.999619405229

In [53]: i ** 2 * 10000
Out[53]: 8410000
```

结果显示，提取随机数的个数越多，统计数值就“莫名其妙”地变得越好[1]。但即使在较大的样本（超过 800 万个随机数）中，统计数值也不等于理想的数字。

幸运的是，可以使用易于实现的通用方差缩减方法来改善（标准）正态分布前两个统计矩的匹配。第一种技术是使用对偶变量。这种方法只提取理想数量一半的随机数，并在之后加入同一组随机数的相反数[2]。例如，如果随机数生成器（即对应的 Python 函

1　这里使用的方法的灵感来自大数法则。——原注

2　这里描述的方法只适用于对称中位数为 0 的随机变量，比如标准正态分布随机变量，我们在本书自始至终几乎只使用这种变量。——原注

数）提取了 0.5，则在数据集中加入另一个值−0.5。从构造上说，这种数据集的均值必然为 0。

用 NumPy 的函数 `np.concatenate()`可以简洁地实现上述方法，下面用对偶变量重复之前的练习：

```
In [54]: sn = npr.standard_normal(int(10000 / 2))
         sn = np.concatenate((sn, -sn)) ❶

In [55]: np.shape(sn) ❷
Out[55]: (10000,)

In [56]: sn.mean() ❸
Out[56]: 2.842170943040401e-18

In [57]: print('%15s %15s' % ('Mean', 'Std. Deviation'))
         print(31 * "-")
         for i in range(1, 31, 2):
             npr.seed(1000)
             sn = npr.standard_normal(i ** 2 * int(10000 / 2))
             sn = np.concatenate((sn, -sn))
             print("%15.12f %15.12f" % (sn.mean(), sn.std()))
                    Mean  Std. Deviation
         -------------------------------
          0.000000000000  1.009653753942
         -0.000000000000  1.000413716783
          0.000000000000  1.002925061201
         -0.000000000000  1.000755212673
          0.000000000000  1.001636910076
         -0.000000000000  1.000726758438
         -0.000000000000  1.001621265149
          0.000000000000  1.001203722778
         -0.000000000000  1.000556669784
         -0.000000000000  1.000113464185
         -0.000000000000  0.999435175324
         -0.000000000000  0.999356961431
         -0.000000000000  0.999641436845
         -0.000000000000  0.999642768905
         -0.000000000000  0.999638303451
```

❶ 连接两个 `ndarray` 对象。

❷ 得到所需的随机数数量。

❸ 结果均值为 0（在标准浮点算术误差之内）。

你立刻就会注意到，这种方法完美地更正了第一个统计矩——这并不令人惊讶，原因

在于每当提取数 n 时，就加入$-n$。因为有了这样的配对，所以整组随机数的均值就会等于 0。然而，这种方法对第二个统计矩——标准差没有任何影响。使用另一种方差缩减技术——矩匹配，有助于在一个步骤中更正第一个和第二个统计矩：

```
In [58]: sn = npr.standard_normal(10000)

In [59]: sn.mean()
Out[59]: -0.001165998295162494

In [60]: sn.std()
Out[60]: 0.991255920204605

In [61]: sn_new = (sn - sn.mean()) / sn.std() ❶

In [62]: sn_new.mean()
Out[62]: -2.3803181647963357e-17

In [63]: sn_new.std()
Out[63]: 0.9999999999999999
```

❶ 在一个步骤中校正第一个和第二个统计矩。

从每个随机数中减去均值并用每个随机数除以标准差，就可以得到一组匹配随机数，（几乎）完美地匹配理想的标准正态分布的第一和第二统计矩。

下面的函数利用对方差缩减技术的认识，用两种、一种或者不用方差缩减技术生成用于过程模拟的标准正态随机数：

```
In [64]: def gen_sn(M, I, anti_paths=True, mo_match=True):
             ''' Function to generate random numbers for simulation.

             Parameters
             ==========
             M: int
                 number of time intervals for discretization
             I: int
                 number of paths to be simulated
             anti_paths: boolean
                 use of antithetic variates
             mo_math: boolean
                 use of moment matching
             '''
             if anti_paths is True:
                sn = npr.standard_normal((M + 1, int(I / 2)))
                sn = np.concatenate((sn, -sn), axis=1)
             else:
```

```
    sn = npr.standard_normal((M + 1, I))
  if mo_match is True:
    sn = (sn - sn.mean()) / sn.std()
  return sn
```

向量化与模拟

NumPy 的向量化是实现 Python 蒙特卡洛模拟算法的一种自然、简洁且高效的方法。不过，使用 NumPy 向量化通常会造成更大的内存占用。同样快速的替代方法请参见第 10 章。

12.3　估值

蒙特卡洛模拟的最重要应用之一是未定权益（期权、衍生品、混合型工具等）的估值。简单地说，在风险中立的世界中，未定权益的价值是风险中立（鞅）测度下的折现后预期收益。这是所有风险因素（股票、指数等）偏离无风险短期利率的概率测度。根据资产定价基本定理，这种概率测度的存在等价于套利机会的缺失。

金融期权表示在规定（行权期）日期（欧式期权）或者规定时期（美式期权）内，以规定价格（所谓行权价）购买（看涨期权）或者出售（看跌期权）指定金融工具。我们首先考虑估值较为简单的情况——欧式期权。

12.3.1　欧式期权

基于某种指数的欧式看涨期权到期日收益可通过公式 $h(S_T) \equiv \max(S_T - K, 0)$ 得出，其中 S_T 是到期日 T 的指数水平，K 是行权价格。若给定相关随机过程（例如几何布朗运动）的风险中立测度，或者在一个完备市场中，这种权证的价格由公式 12-10 表示。

公式 12-10　风险中立预期定价

$$C_0 = e^{-rT} E_0^Q(h(S_T)) = e^{-rT} \int_0^\infty h(s)q(s)ds$$

第 11 章简略介绍了如何通过蒙特卡洛模拟计算积分。接下来会使用这种方法并将其应用到公式 12-10 中。公式 12-11 提供了欧式期权的对应蒙特卡洛模拟公式，其中 $\tilde{S}_T^i$ 是到期日的第 i 个模拟指数水平。

公式 12-11　风险中立蒙特卡洛模拟公式

$$\widetilde{C_0} = e^{-rT} \frac{1}{I} \sum_{i=1}^{I} h(\tilde{S}_T^i)$$

现在考虑几何布朗运动的参数化和估值函数 `gbm_mcs_stat()`，该函数仅以行权价格

作为参数。这里只模拟到期日的指数水平。作为参考，考虑行权价 $K = 105$ 的情况：

```
In [65]: S0 = 100.
         r = 0.05
         sigma = 0.25
         T = 1.0
         I = 50000

In [66]: def gbm_mcs_stat(K):
             ''' Valuation of European call option in Black-Scholes-Merton
             by Monte Carlo simulation (of index level at maturity)

             Parameters
             ==========
             K: float
                 (positive) strike price of the option
             Returns
             =======
             C0: float
                 estimated present value of European call option
             '''
             sn = gen_sn(1, I)
             # simulate index level at maturity
             ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T
                          + sigma * math.sqrt(T) * sn[1])
             # calculate payoff at maturity
             hT = np.maximum(ST - K, 0)
             # calculate MCS estimator
             C0 = math.exp(-r * T) * np.mean(hT)
             return C0

In [67]: gbm_mcs_stat(K=105.) ❶
Out[67]: 10.044221852841922
```

❶ 欧式看涨期权的蒙特卡洛估算函数值。

接下来，我们考虑动态模拟方法，除了看涨期权之外还可以模拟欧式看跌期权。函数 `gbm_mcs_dyna()`实现了这一算法。这段代码还比较相同行权价的看涨和看跌期权的价格估算：

```
In [68]: M = 50 ❶

In [69]: def gbm_mcs_dyna(K, option='call'):
             ''' Valuation of European options in Black-Scholes-Merton
             by Monte Carlo simulation (of index level paths)
```

```
    Parameters
    ==========
    K: float
        (positive) strike price of the option
    option : string
        type of the option to be valued ('call', 'put')

    Returns
    =======
    C0: float
        estimated present value of European call option
    '''
    dt = T / M
    # simulation of index level paths
    S = np.zeros((M + 1, I))
    S[0] = S0
    sn = gen_sn(M, I)
    for t in range(1, M + 1):
        S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt
                + sigma * math.sqrt(dt) * sn[t])
    # case-based calculation of payoff
    if option == 'call':
        hT = np.maximum(S[-1] - K, 0)
    else:
        hT = np.maximum(K - S[-1], 0)
    # calculation of MCS estimator
    C0 = math.exp(-r * T) * np.mean(hT)
    return C0

In [70]: gbm_mcs_dyna(K=110., option='call') ❷
Out[70]: 7.950008525028434

In [71]: gbm_mcs_dyna(K=110., option='put') ❸
Out[71]: 12.629934942682004
```

❶ 离散化时间间隔数量。

❷ 欧式看涨期权的蒙特卡洛估算函数值。

❸ 欧式看跌期权的蒙特卡洛估算函数值。

问题是，这些基于模拟的估值方法与 Black-Scholes-Merton 估值公式得出的基准值相比表现如何？为了找出这种差别，我们利用 `bsm_functions.py` 模块（参见 12.5 节）中的 Black-Scholes-Merton 分析性欧式看涨期权定价公式，来生成一定范围行权价的对应期权价值/估算。首先，我们将静态模拟方法的结果与精确的分析值相比：

```
In [72]: from bsm_functions import bsm_call_value

In [73]: stat_res = [] ❶
         dyna_res = [] ❶
         anal_res = [] ❶
         k_list = np.arange(80., 120.1, 5.) ❷
         np.random.seed(100)

In [74]: for K in k_list:
             stat_res.append(gbm_mcs_stat(K)) ❸
             dyna_res.append(gbm_mcs_dyna(K)) ❸
             anal_res.append(bsm_call_value(S0, K, T, r, sigma)) ❸

In [75]: stat_res = np.array(stat_res) ❹
         dyna_res = np.array(dyna_res) ❹
         anal_res = np.array(anal_res) ❹
```

❶ 初始化收集结果的空列表对象。

❷ 创建一个 `ndarray` 对象，包含行权价范围。

❸ 模拟/计算并收集所有行权价的期权价值。

❹ 将列表对象转换为 `ndarray` 对象。

图 12-15 展示了比较的结果。所有估值的差异都小于 1%，正负的差异均有。

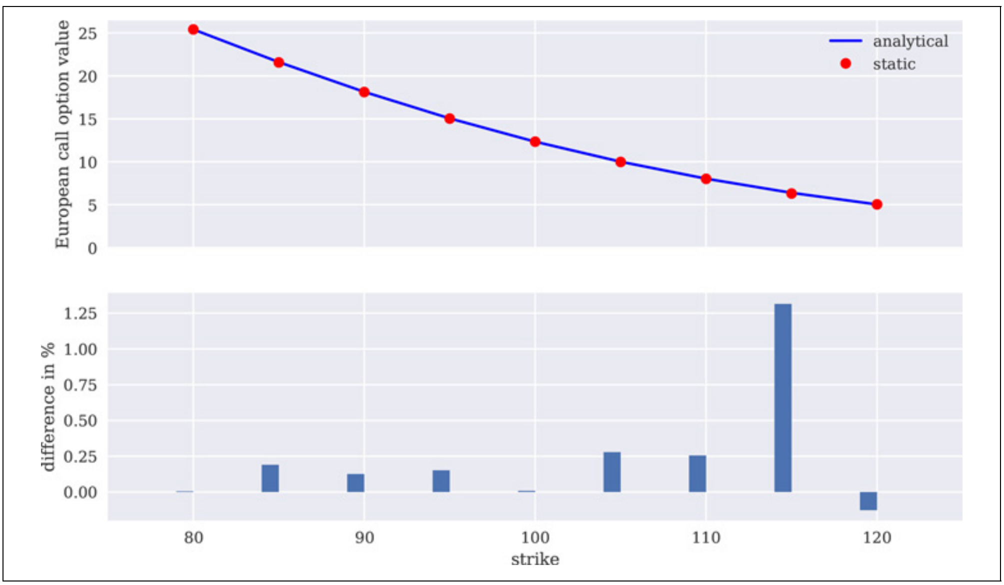


图 12-15　分析性期权价值与蒙特卡洛模拟函数（静态模拟）的对比

```
In [76]: plt.figure(figsize=(10, 6))
         fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(10, 6))
         ax1.plot(k_list, anal_res, 'b', label='analytical')
         ax1.plot(k_list, stat_res, 'ro', label='static')
         ax1.set_ylabel('European call option value')
         ax1.legend(loc=0)
         ax1.set_ylim(bottom=0)
         wi = 1.0
         ax2.bar(k_list - wi / 2, (anal_res - stat_res) / anal_res * 100, wi)
         ax2.set_xlabel('strike')
         ax2.set_ylabel('difference in %')
         ax2.set_xlim(left=75, right=125);
Out[76]: <Figure size 720x432 with 0 Axes>
```

动态模拟和估值方法的情况也类似，结果如图 12-16 所示。同样，所有估值差异小于 1%，标准差既有负数也有正数。作为一般原则，蒙特卡洛估算函数的质量可以通过调整使用的时间间隔 M 和模拟路径数 I 控制：

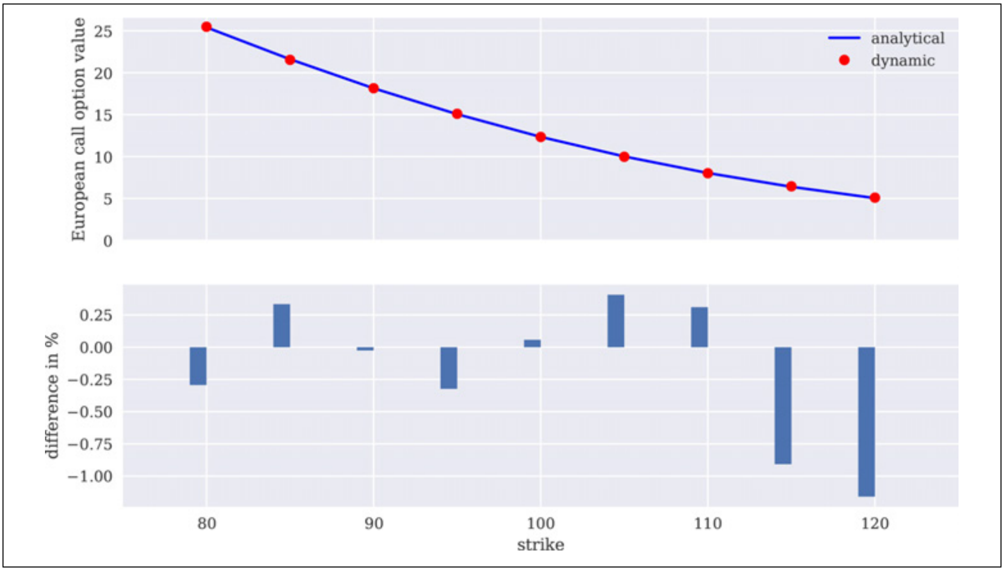


图 12-16　分析性期权价值与蒙特卡洛模拟函数（动态模拟）的对比

```
In [77]: fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(10, 6))
        ax1.plot(k_list, anal_res, 'b', label='analytical')
        ax1.plot(k_list, dyna_res, 'ro', label='dynamic')
        ax1.set_ylabel('European call option value')
        ax1.legend(loc=0)
        ax1.set_ylim(bottom=0)
        wi = 1.0
        ax2.bar(k_list - wi / 2, (anal_res - dyna_res) / anal_res * 100, wi)
```

```
ax2.set_xlabel('strike')
ax2.set_ylabel('difference in %')
ax2.set_xlim(left=75, right=125);
```

12.3.2　美式期权

美式期权的估值比欧式期权的更复杂。在这种情况下，必须解决最优截止问题，然后提出期权的公允价值。公式 12-12 是将美式期权作为最优截止问题时的估值公式。该问题的公式化基于离散的时间网格，以便用于数值化模拟。在某种意义上，更准确地说，这是百慕大式期权的估值公式。时间间隔收敛于 0 长度时，百慕大期权的价值收敛于美式期权的价值。

公式 12-12　以最优截止问题形式出现的美式期权价格

$$V_0 = \sup_{\tau \in \{0, \Delta t, 2\Delta t, \cdots, T\}} \mathrm{e}^{-rT} E_0^Q(h_\tau(S_\tau))$$

我们接下来描述的算法叫作最小二乘蒙特卡洛（LSM）方法，来源于 Longstaff 和 Schwartz（2001）的论文。可以看出，任何给定日期 t 的美式（百慕大）期权价值由 $V_t(s) = \max(h_t(s), C_t(s))$（其中 $C_t(s) = E_t^Q(\mathrm{e}^{-r\Delta t} V_{t+\Delta t}(S_{t+\Delta t}) \mid S_t = s)$ 给出，是给定指数水平 $S_t = s$ 下的期权持续价值。

现在考虑我们在 M 个等长（Δt）的时间间隔中模拟指数水平的 I 条路径。定义 $Y_{t,i} \equiv \mathrm{e}^{-r\Delta t} V_{t+\Delta t,i}$ 为路径 i 在时间 t 时的模拟持续价值。我们不能直接使用这个数字，因为它意味着完美的预期。但是，我们可以使用所有模拟持续价值的截面，通过最小二乘回归估算（预期）持续价值。

给定一组基函数 b_d，$d = 1$，…，D，然后由回归估算公式 $\hat{C}_{t,i} = \sum_{d=1}^{D} \alpha_{d,t}^* \cdot b_d(S_{t,i})$ 算出持续价值，其中最优回归参数 α^*是公式 12-13 中最小二乘问题的解。

公式 12-13　美式期权估值的最小二乘回归

$$\min_{\alpha_{1,t}, \cdots, \alpha_{D,t}} \frac{1}{I} \sum_{i=1}^{I} \left(Y_{t,i} - \sum_{d=1}^{D} \alpha_{d,t} \cdot b_d(S_{t,i}) \right)^2$$

`gbm_mcs_amer()`函数用于实现美式看涨和看跌期权的 LSM 算法[1]：

```
In [78]: def gbm_mcs_amer(K, option='call'):
             ''' Valuation of American option in Black-Scholes-Merton
             by Monte Carlo simulation by LSM algorithm

             Parameters
             ==========
```

1　算法的细节详见 Hilpisch（2015）。——原注

```
        K : float
            (positive) strike price of the option
        option : string
            type of the option to be valued ('call', 'put')

        Returns
        =======
        C0 : float
            estimated present value of European call option
        '''
        dt = T / M
        df = math.exp(-r * dt)
        # simulation of index levels
        S = np.zeros((M + 1, I))
        S[0] = S0
        sn = gen_sn(M, I)
        for t in range(1, M + 1):
            S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt
                    + sigma * math.sqrt(dt) * sn[t])
        # case based calculation of payoff
        if option == 'call':
           h = np.maximum(S - K, 0)
        else:
           h = np.maximum(K - S, 0)
        # LSM algorithm
        V = np.copy(h)
        for t in range(M - 1, 0, -1):
            reg = np.polyfit(S[t], V[t + 1] * df, 7)
            C = np.polyval(reg, S[t])
            V[t] = np.where(C > h[t], V[t + 1] * df, h[t])
        # MCS estimator
        C0 = df * np.mean(V[1])
        return C0

In [79]: gbm_mcs_amer(110., option='call')
Out[79]: 7.721705606305352

In [80]: gbm_mcs_amer(110., option='put')
Out[80]: 13.609997625418051
```

欧式期权的价值处于美式期权价值的下界。两者的差异通常叫作提前行权溢价。接下来，我们比较和以前相同的行权价范围内的欧式和美式期权价值，以估算期权溢价。这次选择看跌期权：[1]

1 由于没有假定任何分红（考虑到指数），看涨期权通常没有提前行权溢价（即没有刺激提前行权的诱因）。——原注

```
In [81]: euro_res = []
         amer_res = []

In [82]: k_list = np.arange(80., 120.1, 5.)

In [83]: for K in k_list:
             euro_res.append(gbm_mcs_dyna(K, 'put'))
             amer_res.append(gbm_mcs_amer(K, 'put'))

In [84]: euro_res = np.array(euro_res)
         amer_res = np.array(amer_res)
```

图 12-17 说明对于所选择的行权价范围，溢价可能最高达到 10%：

```
In [85]: fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(10, 6))
        ax1.plot(k_list, euro_res, 'b', label='European put')
        ax1.plot(k_list, amer_res, 'ro', label='American put')
        ax1.set_ylabel('call option value')
        ax1.legend(loc=0)
        wi = 1.0
        ax2.bar(k_list - wi / 2, (amer_res - euro_res) / euro_res * 100, wi)
        ax2.set_xlabel('strike')
        ax2.set_ylabel('early exercise premium in %')
        ax2.set_xlim(left=75, right=125);
```

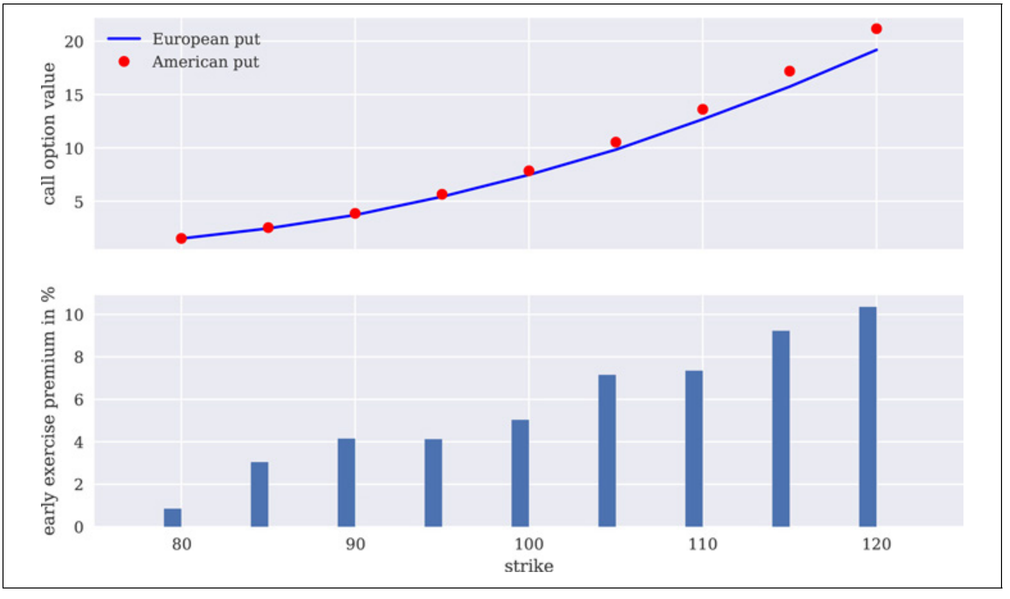


图 12-17　欧式期权和美式期权蒙特卡洛估算值的对比

12.4　风险测度

除了估值之外，风险管理是随机方法和模拟的另一个重要应用领域。本节介绍当今金融行业中非常常用的两个风险测度的计算/估算。

12.4.1　风险价值

风险价值（Value-at-risk，VaR）是最广泛使用的风险测度之一，也是饱受争议的测度之一。从业人员喜欢其直观性，也有许多人对其有限的尾部风险（很快将会详加介绍）捕捉能力进行了广泛的讨论和批评——主要是在理论依据上。从字面上看，VaR 是一个以货币单位（如美元、欧元、日元）表示的数字，表示在给定时间周期中不超过某种置信度（概率）的损失（或者一个投资组合、股票头寸等）。

考虑一个当日价值为 100 万美元的股票头寸，在 30 天（1 个月）内、置信度为 99%的情况下 VaR 为 5 万美元。这个 VaR 数字说明，30 天内损失不超过 5 万美元的概率为 99%（100 个案例中有 99 个）。但是，它并不说明一旦损失超过 5 万美元，损失的规模会达到什么程度；也就是不说明如果最大损失为 10 万或者 50 万美元时，这种特定的“高于 VaR 的损失”概率有多大。它所说明的只是，发生 5 万美元或者更大损失的概率为 1%。

再次假定使用 Black-Scholes-Merton 设置，考虑如下的参数化和未来日期 $T = 30/365$（即 30 天长的一段时期）指数水平的模拟。为了估算 VaR 数字，我们需要模拟的绝对利润和相对于近日持仓价值的亏损，并对其加以排序，排序顺序为从最严重的亏损到最大的利润。图 12-18 展示了模拟绝对绩效的直方图：

```
In [86]: S0 = 100
         r = 0.05
         sigma = 0.25
         T = 30 / 365.
         I = 10000

In [87]: ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T +
                     sigma * np.sqrt(T) * npr.standard_normal(I))  ❶

In [88]: R_gbm = np.sort(ST - S0)  ❷

In [89]: plt.figure(figsize=(10, 6))
         plt.hist(R_gbm, bins=50)
         plt.xlabel('absolute return')
         plt.ylabel('frequency');
```

❶ 模拟集合布朗运动期末值。

❷ 计算每次模拟的绝对利润和损失并排序。

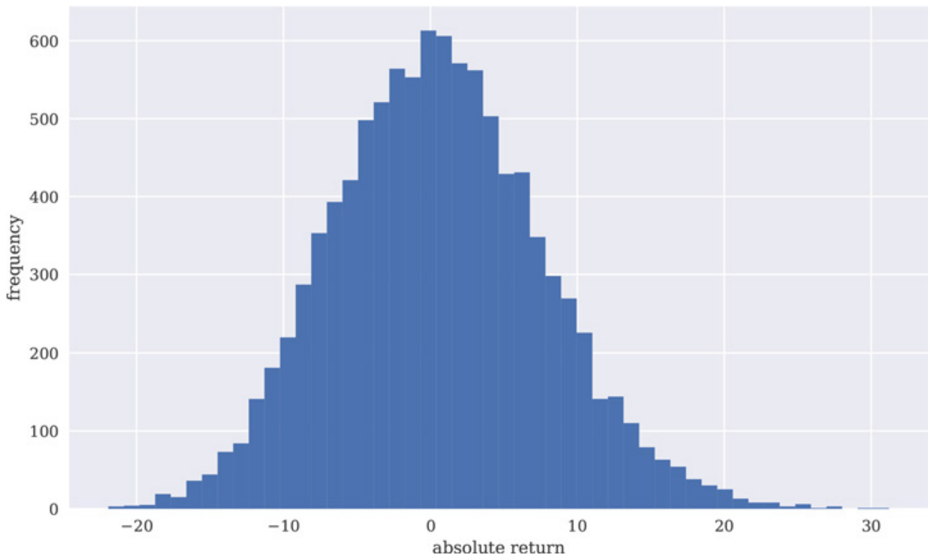


图 12-18　模拟的绝对收益和损失（几何布朗运动）

有了包含排序结果的 `ndarray` 对象后，`scs.scoreatpercentile()`函数已经取得了成功。我们所需要做的就是定义感兴趣的百分比（以百分数表示），在列表对象 `percs` 中，0.1 转换为置信度 100%−0.1%=99.9%。在本例中，置信度为 99.9%的 30 日 VaR 为 18.8 个货币单位，而 90%置信度下为 8.5 个货币单位：

```
In [91]: percs = [0.01, 0.1, 1., 2.5, 5.0, 10.0]
         var = scs.scoreatpercentile(R_gbm, percs)
         print('%16s %16s' % ('Confidence Level', 'Value-at-Risk'))
         print(33 * '-')
         for pair in zip(percs, var):
             print('%16.2f %16.3f' % (100 - pair[0], -pair[1]))
         Confidence Level    Value-at-Risk
         ---------------------------------
                    99.99           21.814
                    99.90           18.837
                    99.00           15.230
                    97.50           12.816
                    95.00           10.824
                    90.00            8.504
```

作为第 2 个例子，回忆一下 Merton 的跳跃扩散，这是一种动态模拟。在这个例子中，利用均值为负数的跳跃成分，我们看到了图 12-19 中类似二项分布的模拟利润/亏损。从正态分布的角度看，左侧有明显的“大尾巴”：

```
In [92]: dt = 30. / 365 / M
         rj = lamb * (math.exp(mu + 0.5 * delta ** 2) - 1)

In [93]: S = np.zeros((M + 1, I))
         S[0] = S0
         sn1 = npr.standard_normal((M + 1, I))
         sn2 = npr.standard_normal((M + 1, I))
         poi = npr.poisson(lamb * dt, (M + 1, I))
         for t in range(1, M + 1, 1):
             S[t] = S[t - 1] * (np.exp((r - rj - 0.5 * sigma ** 2) * dt
                                + sigma * math.sqrt(dt) * sn1[t])
                                + (np.exp(mu + delta * sn2[t]) - 1)
                                * poi[t])
             S[t] = np.maximum(S[t], 0)

In [94]: R_jd = np.sort(S[-1] - S0)

In [95]: plt.figure(figsize=(10, 6))
         plt.hist(R_jd, bins=50)
         plt.xlabel('absolute return')
         plt.ylabel('frequency');
```

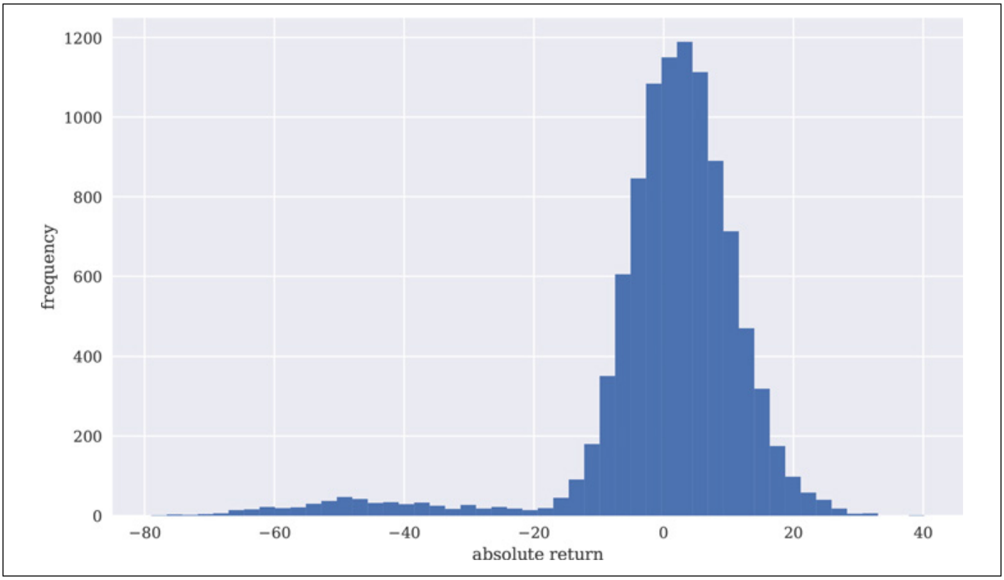


图 12-19　模拟的绝对收益和损失（跳跃扩散）

对于这种过程和参数化，置信度 90%的 30 日 VaR 几乎完全相同，但是在 99.9%置信度的 VaR 与几何布朗运动相比高出 3 倍多（70 对 18.8 个货币单位）：

```
In [96]: percs = [0.01, 0.1, 1., 2.5, 5.0, 10.0]
         var = scs.scoreatpercentile(R_jd, percs)
         print('%16s %16s' % ('Confidence Level', 'Value-at-Risk'))
         print(33 * '-')
         for pair in zip(percs, var):
             print('%16.2f %16.3f' % (100 - pair[0], -pair[1]))
         Confidence Level    Value-at-Risk
         ---------------------------------
                    99.99           76.520
                    99.90           69.396
                    99.00           55.974
                    97.50           46.405
                    95.00           24.198
                    90.00            8.836
```

这展示了标准 VaR 测度捕捉到的在金融市场中经常遇到的尾部风险方面的问题。

为了进一步说明这一点，我们最后以图形方式展示两种情况的 VaR 测度以便比较。如图 12-20 所示，在典型置信度范围内的 VaR 测度表现完全不同：

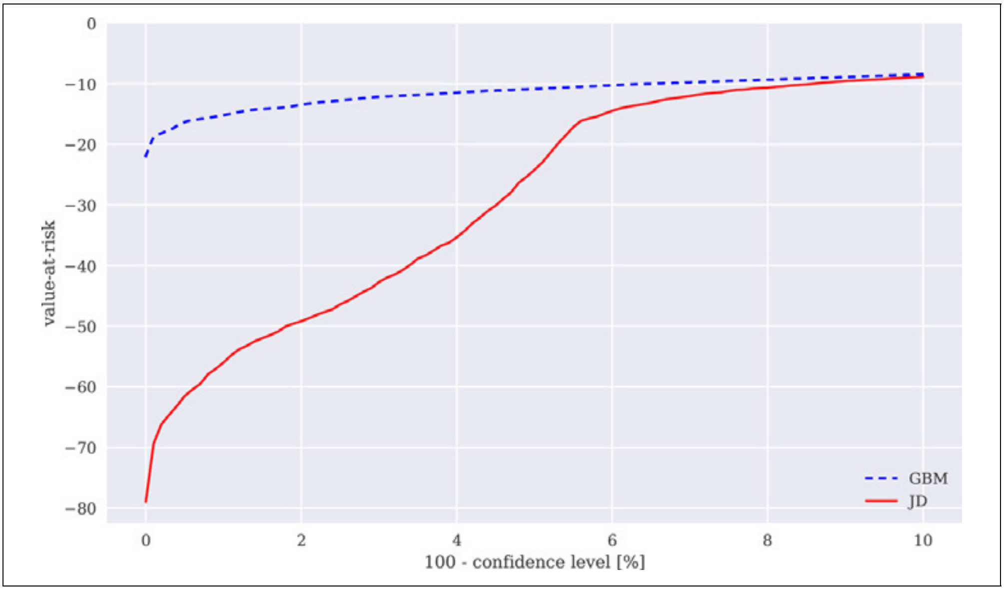


图 12-20　几何布朗运动和跳跃扩散的风险价值

```
In [97]: percs = list(np.arange(0.0, 10.1, 0.1))
         gbm_var = scs.scoreatpercentile(R_gbm, percs)
         jd_var = scs.scoreatpercentile(R_jd, percs)

In [98]: plt.figure(figsize=(10, 6))
```

```
plt.plot(percs, gbm_var, 'b', lw=1.5, label='GBM')
plt.plot(percs, jd_var, 'r', lw=1.5, label='JD')
plt.legend(loc=4)
plt.xlabel('100 - confidence level [%]')
plt.ylabel('value-at-risk')
plt.ylim(ymax=0.0);
```

12.4.2　信用价值调整

其他重要风险测度有信用风险价值（CVaR）和从 CVaR 中派生而来的信用价值调整（CVA）。粗略地讲，CVaR 是对手方可能无法履行其义务所引发风险（例如，对手方破产）的一个测度。在这种情况下，有两个主要的假设：违约概率和（平均）损失水平。

举个具体的例子，再次考虑 Black-Scholes-Merton 的基准设置，并使用如下代码中的参数。在最简单的情况下，可以考虑固定（平均）损失水平 L 和对手方违约（每年）概率 p。使用泊松分布，违约的方案可以用以下代码生成，这里考虑了违约只能发生一次的事实：

```
In [99]: S0 = 100.
         r = 0.05
         sigma = 0.2
         T = 1.
         I = 100000

In [100]: ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T
                       + sigma * np.sqrt(T) * npr.standard_normal(I))

In [101]: L = 0.5 ❶

In [102]: p = 0.01 ❷

In [103]: D = npr.poisson(p * T, I) ❸

In [104]: D = np.where(D > 1, 1, D) ❹
```

❶ 定义损失水平。

❷ 定义违约概率。

❸ 模拟违约事件。

❹ 将违约限制为 1 次。

如果没有违约，未来指数水平的风险中立价值应该等于资产当日现值（取决于数值误差造成的差异）。CVaR 和经过信用风险调整之后的资产现值可以这样计算：

```
In [105]: math.exp(-r * T) * np.mean(ST) ❶
Out[105]: 99.94767178982691

In [106]: CVaR = math.exp(-r * T) * np.mean(L * D * ST) ❷
          CvaR ❷
Out[106]: 0.4883560258963962

In [107]: S0_CVA = math.exp(-r * T) * np.mean((1 - L * D) * ST) ❸
          S0_CVA ❸
Out[107]: 99.45931576393053

In [108]: S0_adj = S0 - CVaR ❹
          S0_adj ❹
Out[108]: 99.5116439741036
```

❶ T 日的贴现平均模拟资产价值。

❷ CVaR 是违约情况下的未来损失贴现平均值。

❸ 模拟违约损失调整后的 T 日贴现平均模拟资产价值。

❹ 模拟 CVaR 调整后的当前资产价格。

在这个特殊的模拟示例中，我们观察到由于信用风险引起的大约 1000 次亏损，这是假定违约概率为 1%、10 万次模拟下预期的结果。图 12-21 展示了由于违约引起亏损的完整频率分布。当然，在大部分情况下（10 万例中的 99 000 例），没有发现亏损：

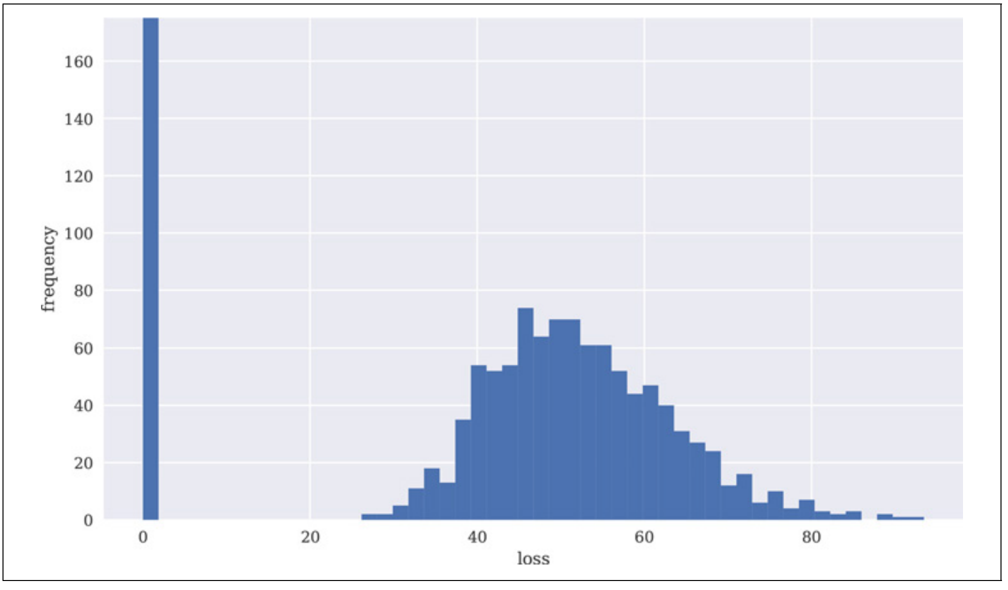


图 12-21　由于风险中立预期违约引起的亏损（股票）

```
In [109]: np.count_nonzero(L * D * ST) ❶
Out[109]: 978

In [110]: plt.figure(figsize=(10, 6))
          plt.hist(L * D * ST, bins=50)
          plt.xlabel('loss')
          plt.ylabel('frequency')
          plt.ylim(ymax=175);
```

❶ 违约事件以及因此引起的损失事件数量。

现在考虑欧式看涨期权的情况。它在行权价 100 时的价值大约为 10.4 个货币单位。在相同的违约概率和损失水平假设下，CVaR 大约为 5 分：

```
In [111]: K = 100.
          hT = np.maximum(ST - K, 0)

In [112]: C0 = math.exp(-r * T) * np.mean(hT) ❶
          C0 ❶
Out[112]: 10.396916492839354

In [113]: CVaR = math.exp(-r * T) * np.mean(L * D * hT) ❷
          CVaR ❷
Out[113]: 0.05159099858923533

In [114]: C0_CVA = math.exp(-r * T) * np.mean((1 - L * D) * hT) ❸
          C0_CVA ❸
Out[114]: 10.34532549425012
```

❶ 欧式看涨期权的蒙特卡洛估算值。

❷ CVaR 是违约情况下的未来损失贴现平均值。

❸ 欧式看涨期权的蒙特卡洛估算值，经模拟的违约损失调整。

和常规资产相比，期权有不同的特性。我们只看到略低于 500 次因违约引起的亏损，但是仍然有大约 1000 次违约。这一结果源于这样的事实：期权到期日时的收益为 0 的概率很大。图 12-22 说明，期权的 CVaR 和常规资产相比有着完全不同的频率分布：

```
In [115]: np.count_nonzero(L * D * hT) ❶
Out[115]: 538

In [116]: np.count_nonzero(D) ❷
Out[116]: 978

In [117]: I - np.count_nonzero(hT) ❸
Out[117]: 44123
```

```
In [118]: plt.figure(figsize=(10, 6))
          plt.hist(L * D * hT, bins=50)
          plt.xlabel('loss')
          plt.ylabel('frequency')
          plt.ylim(ymax=350);
```

❶ 由于违约造成损失的次数。

❷ 违约次数。

❸ 到期时期权无价值的次数。

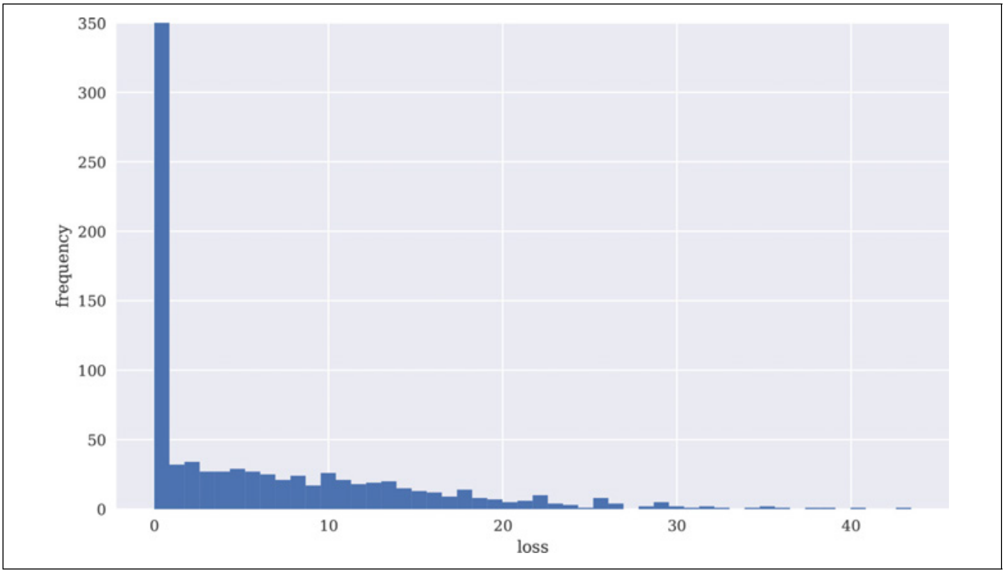


图 12-22　由于风险中立预期违约引起的亏损（看涨期权）

12.5　Python 脚本

下面介绍与 Black-Scholes-Merton 模型相关的核心函数实现，这些函数用于欧式（看涨）期权的分析定价。模型的细节参见 Black 和 Scholes（1973）以及 Merton（1973）的著作。附录 B 有基于 Python 类的另一个实现。

```
#
# Valuation of European call options
# in Black-Scholes-Merton model
# incl. vega function and implied volatility estimation
# bsm_functions.py
#
```

```
# (c) Dr. Yves J. Hilpisch
# Python for Finance, 2nd ed.
#


def bsm_call_value(S0, K, T, r, sigma):
    ''' Valuation of European call option in BSM model.
    Analytical formula.

    Parameters
    ==========
    S0: float
        initial stock/index level
    K: float
        strike price
    T: float
        maturity date (in year fractions)
    r: float
        constant risk-free short rate
    sigma: float
        volatility factor in diffusion term

    Returns
    =======
    value: float
        present value of the European call option
    '''
    from math import log, sqrt, exp
    from scipy import stats

    S0 = float(S0)
    d1 = (log(S0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * sqrt(T))
    d2 = (log(S0 / K) + (r - 0.5 * sigma ** 2) * T) / (sigma * sqrt(T))
    # stats.norm.cdf --> cumulative distribution function
    #                    for normal distribution
    value = (S0 * stats.norm.cdf(d1, 0.0, 1.0) -
             K * exp(-r * T) * stats.norm.cdf(d2, 0.0, 1.0))
    return value


def bsm_vega(S0, K, T, r, sigma):
    ''' Vega of European option in BSM model.

    Parameters
    ==========
```

```
    S0: float
        initial stock/index level
    K: float
        strike price
    T: float
        maturity date (in year fractions)
    r: float
        constant risk-free short rate
    sigma: float
        volatility factor in diffusion term

    Returns
    =======
    vega: float
        partial derivative of BSM formula with respect
        to sigma, i.e. vega

    '''
    from math import log, sqrt
    from scipy import stats
    S0 = float(S0)
    d1 = (log(S0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * sqrt(T))
    vega = S0 * stats.norm.pdf(d1, 0.0, 1.0) * sqrt(T)
    return vega

# Implied volatility function


def bsm_call_imp_vol(S0, K, T, r, C0, sigma_est, it=100):
    ''' Implied volatility of European call option in BSM model.

    Parameters
    ==========
    S0: float
        initial stock/index level
    K: float
        strike price
    T: float
        maturity date (in year fractions)
    r: float
        constant risk-free short rate
    sigma_est: float
        estimate of impl. volatility
    it: integer
        number of iterations
```

```
    Returns
    =======
    simga_est: float
        numerically estimated implied volatility
    '''
    for i in range(it):
        sigma_est -= ((bsm_call_value(S0, K, T, r, sigma_est) - C0) /
                      bsm_vega(S0, K, T, r, sigma_est))
    return sigma_est
```

12.6 结语

本章介绍对蒙特卡洛模拟在金融学中的应用起重要作用的方法和技术，特别说明了如何根据不同的分布法则生成（伪）随机数。接下来，本章继续介绍了在许多金融领域中很重要的随机变量及随机过程模拟。本章深入地讨论了两个应用领域：欧式和美式期权的估值以及风险价值和信用价值调整等风险测度的估算。

本章说明 Python 和 NumPy 的组合很适合于实现计算要求很高的任务，如通过蒙特卡洛模拟进行的美式期权估值。这主要是因为 NumPy 的大部分函数和类用 C 语言实现，使其在一般情况下对纯 Python 代码有明显的速度优势。由于向量化操作，NumPy 还具有代码紧凑和易于理解的好处。

12.7 延伸阅读

最早在金融学中引入蒙特卡洛模拟的论文是：

- Boyle, Phelim (1977).“Options: A Monte Carlo Approach.”*Journal of Financial Economics*, Vol. 4, No. 4, pp. 322-338.

本章引用的其他原创论文（参见第 18 章）：

- Black, Fischer, and Myron Scholes (1973). “The Pricing of Options and Corporate Liabilities.” *Journal of Political Economy*, Vol. 81, No. 3, pp. 638-659.
- Cox, John, Jonathan Ingersoll, and Stephen Ross (1985). “A Theory of the Term Structure of Interest Rates.” *Econometrica*, Vol. 53, No. 2, pp. 385-407.
- Heston, Steven (1993). “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies*, Vol. 6, No. 2, 327-343.

- Merton, Robert (1973). “Theory of Rational Option Pricing.” *Bell Journal of Economics and Management Science*, Vol. 4, pp. 141-183.
- Merton, Robert (1976). “Option Pricing When the Underlying Stock Returns Are Discontinuous.” *Journal of Financial Economics*, Vol. 3, No. 3, pp. 125-144.

下列图书深入介绍了本章的主题（不过，第一本书没有涵盖技术实现细节）：

- Glasserman, Paul (2004). *Monte Carlo Methods in Financial Engineering*. New York: Springer.
- Hilpisch, Yves (2015). *Derivatives Analytics with Python*. Chichester, England: Wiley Finance.

直到世纪之交，通过蒙特卡洛模拟估算美式期权价值的有效方法才最终发布：

- Longstaff, Francis, and Eduardo Schwartz (2001). “Valuing American Options by Simulation: A Simple Least Squares Approach.” *Review of Financial Studies*, Vol. 14, No. 1, pp. 113-147.

下面的图书广泛、深入地介绍了信用风险的处置方法：

- Duffie, Darrell, and Kenneth Singleton (2003). *Credit Risk—Pricing, Measurement, and Management*. Princeton, NJ: Princeton University Press.

第 13 章

统计学

我可以用统计学证明一切，除了真相。

——乔治·坎宁

统计学是一个广泛的领域。该领域提供的工具和结果已经成为金融学中不可或缺的一部分。这也就能够解释为什么 R 等领域专用语言在金融行业中流行。统计模型越精细、越复杂，易于使用和高性能的计算解决方案就越重要。

只用本书一章的篇幅无法充分说明统计领域的丰富和广博。因此，如其他章一样，我们将重点放在精选的主题上，这些主题极其重要，能够为使用 Python 完成特定任务提供好的起点。本章有以下 4 个重点。

正态性检验

许多重要金融模型，如均值-方差投资组合理论和资本性资产定价模型（CAPM），都依赖于证券收益呈正态分布这一假设。因此，本章介绍一些方法，测试给定时间序列的正态性。

投资组合优化

可以将现代投资组合理论（MPT）看作金融统计学的最大成功之一。从 20 世纪 50 年代初开始，由于先驱者 Harry Markowitz 的努力，这一理论开始用严格的数学和统计学方法，代替人们投资金融市场时对判断和经验的依赖。在这个意义上，它可能是金融学中第一种真正的计量方法。

贝叶斯统计

从概念层面上，贝叶斯统计在统计学中引入了主体信念度和信念度更新；例如，对于线性回归，可能采取回归参数统计分布的形式代替单点估算（如回归线截距

和斜率）。如今，贝叶斯方法在金融学中相当流行和重要，这也是我们在本章中阐述一些（高级）应用的原因。

机器学习

机器学习（或者统计学习）基于高级统计学方法，被视为人工智能（AI）学科的分支。和统计学本身类似，机器学习提供一组丰富的方法和模型，以便从数据集中学习，并根据所学建立预测。不同的学习算法有很大的区别，例如有监督学习或无监督学习。算法解决的问题类型也不同，如估算或者分类。本章介绍的例子属于有监督学习分类。

本章在许多方面上与日期或时间信息有关。用 Python、NumPy 和 pandas 处理这种数据的概述参见附录 A。

13.1 正态性检验

可以说，正态分布是金融学中最重要的分布之一，也是金融理论的主要统计学基础之一。尤其是下面这些金融理论基础，在很大程度上依赖于股票市场收益的正态分布[1]。

投资组合理论

当股票收益呈正态分布时，最优化投资组合可以在这样的环境中选择：只有平均收益、收益的方差（或者波动率）以及不同股票之间的协方差与投资决策（即最优化投资组合构成）相关。

资本性资产定价模型

同样，当股票收益呈正态分布时，单独证券的价格可以很好地用和某种大规模市场指数的关系表示；这种关系通常用单一股票与市场指数的联动指标（β）表示。

有效市场假设

有效市场指的是价格反映所有可用信息的市场，其中的“所有”可以是狭义的，也可以是广义的（例如“所有公开信息”或者同时包括“只为个人所有”的信息）。如果这个假设成立，股票价格波动将是随机的，而收益呈正态分布。

期权定价理论

布朗运动是随机股票（和其他证券）价格变动的标准、基准模型。著名的 Black-Scholes-Merton 期权定价公式使用几何布朗运动作为股票在一段时间内随机波动

1 另一个核心假设是线性假设。例如，一般来说，我们假设金融市场的需求（如股份数量）与（购买股份的）价格呈线性关系。换言之，我们一般假设市场是完全流动的，变化的需求对金融工具的单价没有影响。——原注

的模型，这种波动造成收益呈正态分布。

上述的理论只是支持金融学中正态性假设重要性的一部分原因。

13.1.1 基准案例

为了给进一步的分析提供基础，我们从几何布朗运动开始介绍，它是金融建模使用的规范随机过程之一。关于几何布朗运动中的路径特性，说明如下。

正态对数收益率

在两个时间点之间的对数收益率 $\log \frac{S_t}{S_s} = \log S_t - \log S_s$（$0 < s < t$）呈正态分布。

对数-正态价值

在任何时间点 $t > 0$，价值 S_t 呈正态分布。

为了以后工作，首先需要进行绘图设置。此外，我们需要一些 Python 库，包括 `scipy.stats` 和 `statsmodels.api`：

```
In [1]: import math
        import numpy as np
        import scipy.stats as scs
        import statsmodels.api as sm
        from pylab import mpl, plt

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

使用函数 `gen_paths()` 为几何布朗运动生成蒙特卡洛路径（参见第 12 章）：

```
In [3]: def gen_paths(S0, r, sigma, T, M, I):
            ''' Generate Monte Carlo paths for geometric Brownian motion.

            Parameters
            ==========
            S0: float
                initial stock/index value
            r: float
                constant short rate
            sigma: float
                constant volatility
            T: float
                final time horizon
            M: int
                number of time steps/intervals
```

```
    I: int
        number of paths to be simulated

    Returns
    =======
    paths: ndarray, shape (M + 1, I)
        simulated paths given the parameters
    '''
    dt = T / M
    paths = np.zeros((M + 1, I))
    paths[0] = S0
    for t in range(1, M + 1):
        rand = np.random.standard_normal(I)
        rand = (rand - rand.mean()) / rand.std() ❶
        paths[t] = paths[t - 1] * np.exp((r - 0.5 * sigma **
       2) * dt + sigma * math.sqrt(dt) * rand) ❷
    return paths
```

❶ 匹配一阶矩和二阶矩。

❷ 向量化的几何布朗运动欧拉离散。

模拟基于以下所示的蒙特卡洛模拟参数。它与 `gen_paths()`函数相结合，生成 25 万条路径，每条路径有 50 个时间步。图 13-1 展示了前 10 条模拟路径：

```
In [4]: S0 = 100. ❶
        r = 0.05 ❷
        sigma = 0.2 ❸
        T = 1.0 ❹
        M = 50 ❺
        I = 250000 ❻
        np.random.seed(1000)

In [5]: paths = gen_paths(S0, r, sigma, T, M, I)

In [6]: S0 * math.exp(r * T) ❼

Out[6]: 105.12710963760242

In [7]: paths[-1].mean() ❼
Out[7]: 105.12645392478755

In [8]: plt.figure(figsize=(10, 6))
        plt.plot(paths[:, :10])
        plt.xlabel('time steps')
        plt.ylabel('index level');
```

❶ 模拟过程初始值。

❷ 恒定短期利率。

❸ 恒定波动率因子。

❹ 以年表示的时间范围。

❺ 时间间隔数量。

❻ 模拟过程数量。

❼ 预期价值和平均模拟值。



图 13-1　几何布朗运动的模拟路径

我们感兴趣的是主要对数收益率的分布。下面的代码生成一个包含所有对数收益率的 ndarray 对象，它基于模拟路径。这里展示的是一条模拟路径和结果对数收益率：

```
In [9]: paths[:, 0].round(4)
Out[9]: array([100.    ,  97.821 ,  98.5573, 106.1546, 105.899 ,  99.8363,
                100.0145, 102.6589, 105.6643, 107.1107, 108.7943, 108.2449,
                106.4105, 101.0575, 102.0197, 102.6052, 109.6419, 109.5725,
                112.9766, 113.0225, 112.5476, 114.5585, 109.942 , 112.6271,
                112.7502, 116.3453, 115.0443, 113.9586, 115.8831, 117.3705,
                117.9185, 110.5539, 109.9687, 104.9957, 108.0679, 105.7822,
                105.1585, 104.3304, 108.4387, 105.5963, 108.866 , 108.3284,
                107.0077, 106.0034, 104.3964, 101.0637,  98.3776,  97.135 ,
                 95.4254,  96.4271,  96.3386])
```

```
In [10]: log_returns = np.log(paths[1:] / paths[:-1])

In [11]: log_returns[:, 0].round(4)
Out[11]: array([-0.022 ,  0.0075,  0.0743, -0.0024, -0.059 ,  0.0018, 0.0261,
                 0.0289,  0.0136,  0.0156, -0.0051, -0.0171, -0.0516, 0.0095,
                 0.0057,  0.0663, -0.0006,  0.0306,  0.0004, -0.0042, 0.0177,
                -0.0411,  0.0241,  0.0011,  0.0314, -0.0112, -0.0095, 0.0167,
                 0.0128,  0.0047, -0.0645, -0.0053, -0.0463,  0.0288,-0.0214,
                -0.0059, -0.0079,  0.0386, -0.0266,  0.0305, -0.0049,-0.0123,
                -0.0094, -0.0153, -0.0324, -0.0269, -0.0127, -0.0178, 0.0104,
                -0.0009])
```

这是人们在金融市场上可能经历的：在一些日子里你的投资获得正收益，而在其他日子里，相对于最近的财富状况，你损失了金钱。

`print_statistics()`函数是`scipy.stats`子库中的`scs.describe()`函数的包装器函数。它主要以更易于（人类）理解的方式输出给定（历史或者模拟）数据集均值、偏斜度或者峰度等统计量：

```
In [13]: def print_statistics(array):
             ''' Prints selected statistics.

             Parameters
             ==========
             array: ndarray
                 object to generate statistics on
             '''
             sta = scs.describe(array)
             print('%14s %15s' % ('statistic', 'value'))
             print(30 * '-')
             print('%14s %15.5f' % ('size', sta[0]))
             print('%14s %15.5f' % ('min', sta[1][0]))
             print('%14s %15.5f' % ('max', sta[1][1]))
             print('%14s %15.5f' % ('mean', sta[2]))
             print('%14s %15.5f' % ('std', np.sqrt(sta[3])))
             print('%14s %15.5f' % ('skew', sta[4]))
             print('%14s %15.5f' % ('kurtosis', sta[5]))

In [14]: print_statistics(log_returns.flatten())
              statistic           value
         ------------------------------
                   size  12500000.00000
                    min        -0.15664
                    max         0.15371
                   mean         0.00060
```

```
                std         0.02828
               skew         0.00055
           kurtosis         0.00085

In [15]: log_returns.mean() * M + 0.5 * sigma ** 2 ❶
Out[15]: 0.05000000000000005

In [16]: log_returns.std() * math.sqrt(M) ❷
Out[16]: 0.20000000000000015
```

❶ 按照伊藤公式校正的年化对数收益率均值。

❷ 年化波动率，也就是对数收益的年化标准差。

本例中的数据集包含 1250 万个数据点，其值主要处于−0.15 和 0.15 之间。我们可以假设预期平均年化收益为 0.05，标准差（波动率）为 0.2。数据集中的年化值不完全等于上述值，但是很接近（均值乘以 50，标准差乘以 $\sqrt{50}$ ）。匹配良好的原因之一是在提取随机数时使用矩匹配来减小方差（参见 12.2.3 节）。

图 13-2 是对模拟对数收益率的频率分布以及参数化 `r` 和 `sigma` 之后的正态分布概率密度函数（PDF）的对比。使用的函数是 `scipy.stats` 子库中的 `norm.pdf()`。两者明显很相符：

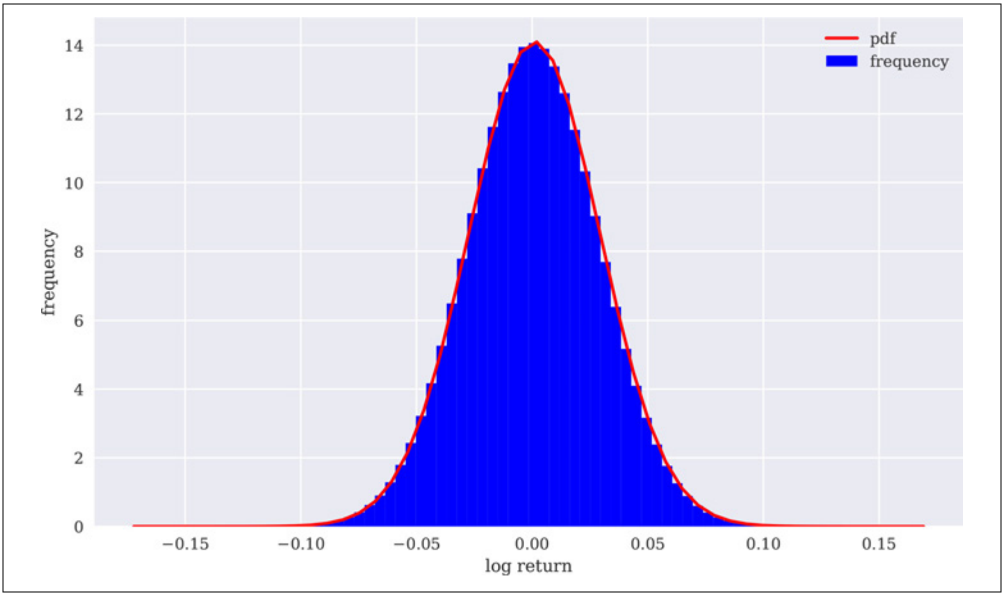


图 13-2　对数收益率和正态密度函数的直方图

```
In [17]: plt.figure(figsize=(10, 6))
         plt.hist(log_returns.flatten(), bins=70, normed=True,
```

```
                label='frequency', color='b')
        plt.xlabel('log return')
        plt.ylabel('frequency')
        x = np.linspace(plt.axis()[0], plt.axis()[1])
        plt.plot(x, scs.norm.pdf(x, loc=r / M, scale=sigma / np.sqrt(M)),
                 'r', lw=2.0, label='pdf') ❶
        plt.legend();
```

❶ 绘制假定参数（按照时间间隔长度调整）的 PDF 图表。

对比频率分布（直方图）与理论化 PDF 不是图形化“检验”正态性的唯一方法。所谓的分位数-分位数（quantile-quantile，QQ）图也很适合于这一任务。在这种图表中，我们对样本分位数值与理论分位数值进行对比。对于正态分布的样本数据集，QQ 图如图 13-3 所示，绝大多数分位数值（点）位于一条直线上：

```
In [18]: sm.qqplot(log_returns.flatten()[::500], line='s')
         plt.xlabel('theoretical quantiles')
         plt.ylabel('sample quantiles');
```

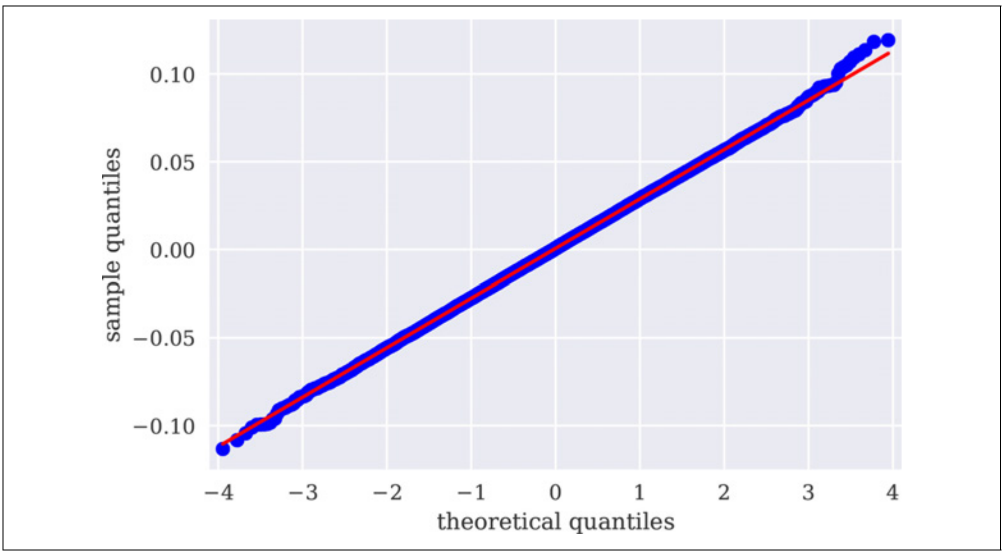


图 13-3　几何布朗运动对数收益率的分位数-分位数图

尽管图形方法很有吸引力，但是它们通常无法代替更严格的测试过程。`normality_tests()` 函数组合了 3 种不同的统计学测试，如下所示。

偏斜度测试（*`skewtest()`*）

测试样本数据的偏斜是否“正态”（也就是值足够接近 0）。

峰度测试（**`kurtosistest()`**）

　　与偏斜度测试类似，测试样本数据的峰度是否“正态”（同样是值足够接近 0）。

正态性测试（**`normaltest()`**）

　　组合其他两种测试方法，检验正态性。

测试值表明，几何布朗运动的对数收益率确实呈正态分布。也就是说，它们的 p 值为 0.05 或者更高：

```
In [19]: def normality_tests(arr):
             ''' Tests for normality distribution of given data set.

             Parameters
             ==========
             array: ndarray
                 object to generate statistics on
             '''
             print('Skew of data set %14.3f' % scs.skew(arr))
             print('Skew test p-value %14.3f' % scs.skewtest(arr)[1])
             print('Kurt of data set %14.3f' % scs.kurtosis(arr))
             print('Kurt test p-value %14.3f' % scs.kurtosistest(arr)[1])
             print('Norm test p-value %14.3f' % scs.normaltest(arr)[1])

In [20]: normality_tests(log_returns.flatten()) ❶
         Skew of data set           0.001
         Skew test p-value          0.430
         Kurt of data set           0.001
         Kurt test p-value          0.541
         Norm test p-value          0.607
```

❶ 所有 p 值都明显高于 0.05。

最后，我们检查期末值是否呈正态分布。这也归结于正态性检验，因为我们只需要应用对数函数来转换数据以得到正态分布数据（也可能得不到）。图 13-4 中绘制了正态分布的期末值和转换后的值（“对数指数水平”）：

```
In [21]: f, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 6))
         ax1.hist(paths[-1], bins=30)
         ax1.set_xlabel('index level')
         ax1.set_ylabel('frequency')
         ax1.set_title('regular data')
         ax2.hist(np.log(paths[-1]), bins=30)
         ax2.set_xlabel('log index level')
         ax2.set_title('log data')
```

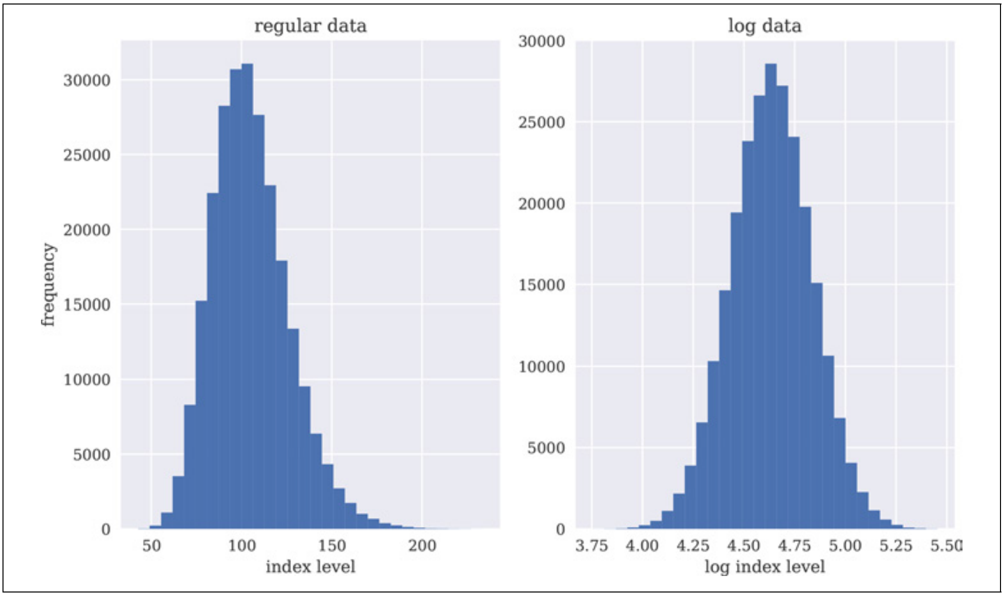


图 13-4　几何布朗运动的模拟期末指数水平直方图

数据集的统计量和预期的表现一样——例如，均值接近 105。对数指数水平值的偏斜度和峰度接近于 0，且 p 值很高，这表示强烈支持正态分布假设：

```
In [22]: print_statistics(paths[-1])
              statistic           value
         ------------------------------
                   size    250000.00000
                    min        42.74870
                    max       233.58435
                   mean       105.12645
                    std        21.23174
                   skew         0.61116
               kurtosis         0.65182

In [23]: print_statistics(np.log(paths[-1]))
              statistic           value
         ------------------------------
                   size    250000.00000
                    min         3.75534
                    max         5.45354
                   mean         4.63517
                    std         0.19998
                   skew        -0.00092
               kurtosis        -0.00327
```

```
In [24]: normality_tests(np.log(paths[-1]))
         Skew of data set          -0.001
         Skew test p-value          0.851
         Kurt of data set          -0.003
         Kurt test p-value          0.744
         Norm test p-value          0.931
```

图 13-5 再次比较频率分布和正态分布的 PDF，两者相当匹配（当然，这在意料之中）：

```
In [25]: plt.figure(figsize=(10, 6))
         log_data = np.log(paths[-1])
         plt.hist(log_data, bins=70, normed=True,
                  label='observed', color='b')
         plt.xlabel('index levels')
         plt.ylabel('frequency')
         x = np.linspace(plt.axis()[0], plt.axis()[1])
         plt.plot(x, scs.norm.pdf(x, log_data.mean(), log_data.std()),
                  'r', lw=2.0, label='pdf')
         plt.legend();
```

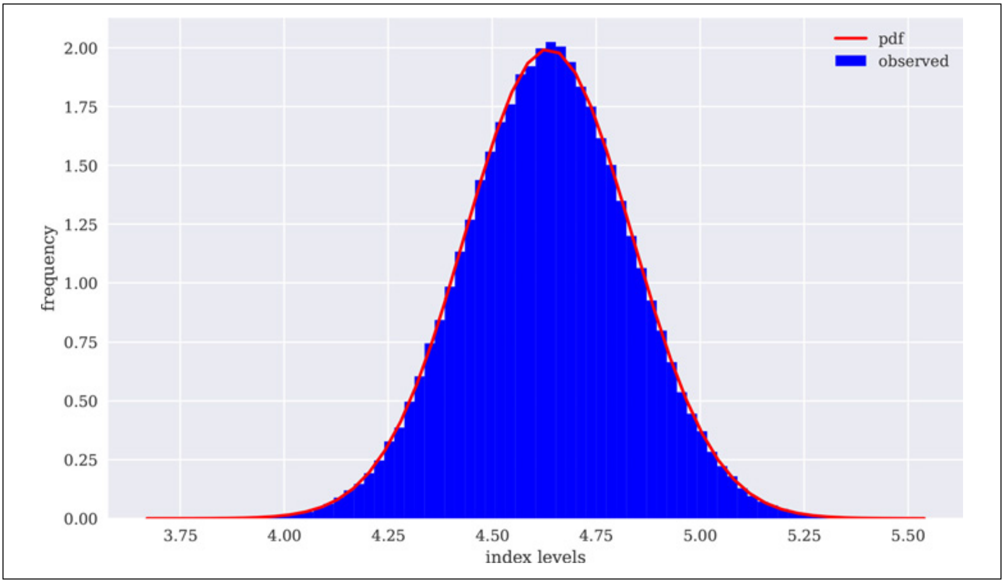


图 13-5　几何布朗运动对数指数水平和正态密度函数的直方图

图 13-6 表示也支持对数指数水平呈正态分布的假设：

```
In [26]: sm.qqplot(log_data, line='s')
         plt.xlabel('theoretical quantiles')
         plt.ylabel('sample quantiles');
```

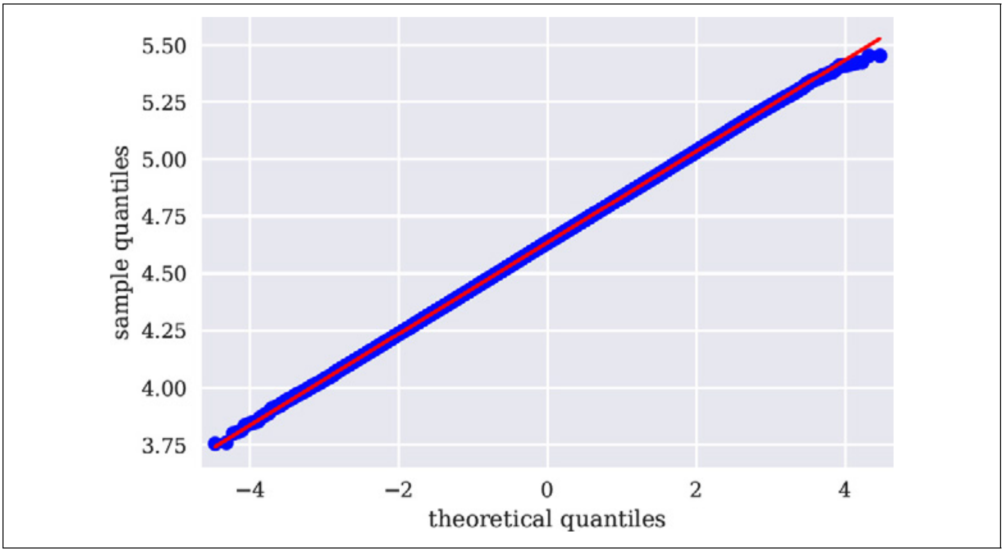


图 13-6　几何布朗运动对数指数水平的分位数-分位数图



正态性

证券收益的正态性假设是许多重要金融理论的核心。Python 提供了高效的统计和图形手段，以测试时间序列数据是否为正态分布。

13.1.2　真实数据

本节将分析 4 个历史金融时间序列，其中两个是技术型公司的股票，另两个是交易型开放式基金（ETF），具体如下。

- APPL.0：苹果公司股价。
- MSFT.0：微软公司股价。
- SPY：标准普尔 500 ETF。
- GLD：标准普尔黄金 ETF。

我们选择的数据管理工具是 pandas（参见第 8 章），图 13-7 展示了一段时间中的规格化价格：

```
In [27]: import pandas as pd

In [28]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                           index_col=0, parse_dates=True).dropna()

In [29]: symbols = ['SPY', 'GLD', 'AAPL.O', 'MSFT.O']
```

```
In [30]: data = raw[symbols]
         data = data.dropna()

In [31]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2138 entries, 2010-01-04 to 2018-06-29
         Data columns (total 4 columns):
         SPY       2138 non-null float64
         GLD       2138 non-null float64
         AAPL.O    2138 non-null float64
         MSFT.O    2138 non-null float64
         dtypes: float64(4)
         memory usage: 83.5 KB

In [32]: data.head()
Out[32]:                SPY     GLD      AAPL.O  MSFT.O
         Date
         2010-01-04  113.33  109.80  30.572827  30.950
         2010-01-05  113.63  109.70  30.625684  30.960
         2010-01-06  113.71  111.51  30.138541  30.770
         2010-01-07  114.19  110.82  30.082827  30.452
         2010-01-08  114.57  111.37  30.282827  30.660

In [33]: (data / data.iloc[0] * 100).plot(figsize=(10, 6))
```

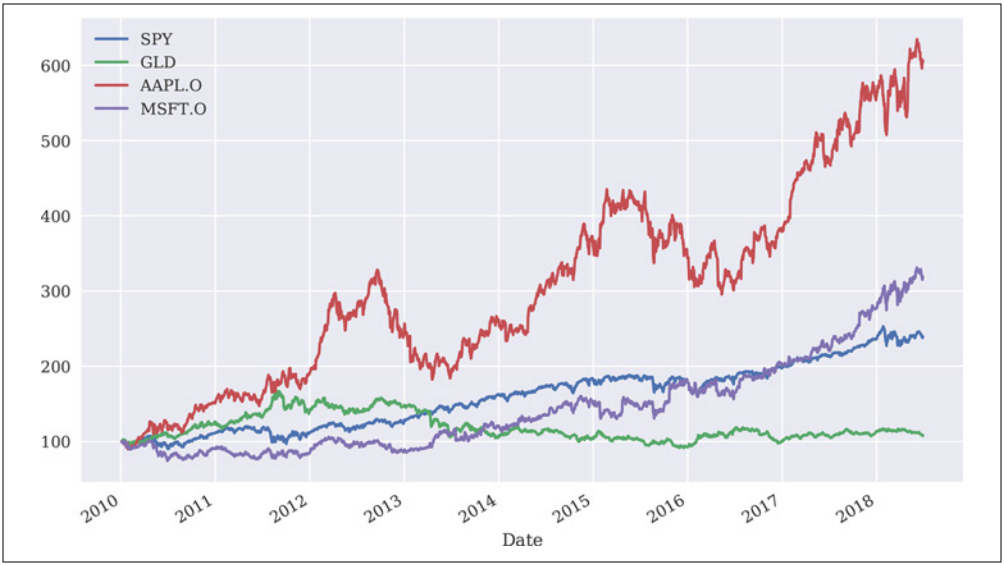


图 13-7　金融工具在一段时间内的规格化价格

图 13-8 展示了金融工具对数回报率的直方图：

```
In [34]: log_returns = np.log(data / data.shift(1))
         log_returns.head()
Out[34]:                   SPY       GLD    AAPL.O     MSFT.O
         Date
         2010-01-04        NaN       NaN       NaN        NaN
         2010-01-05   0.002644 -0.000911  0.001727   0.000323
         2010-01-06   0.000704  0.016365 -0.016034  -0.006156
         2010-01-07   0.004212 -0.006207 -0.001850  -0.010389
         2010-01-08   0.003322  0.004951  0.006626   0.006807

In [35]: log_returns.hist(bins=50, figsize=(10, 8));
```

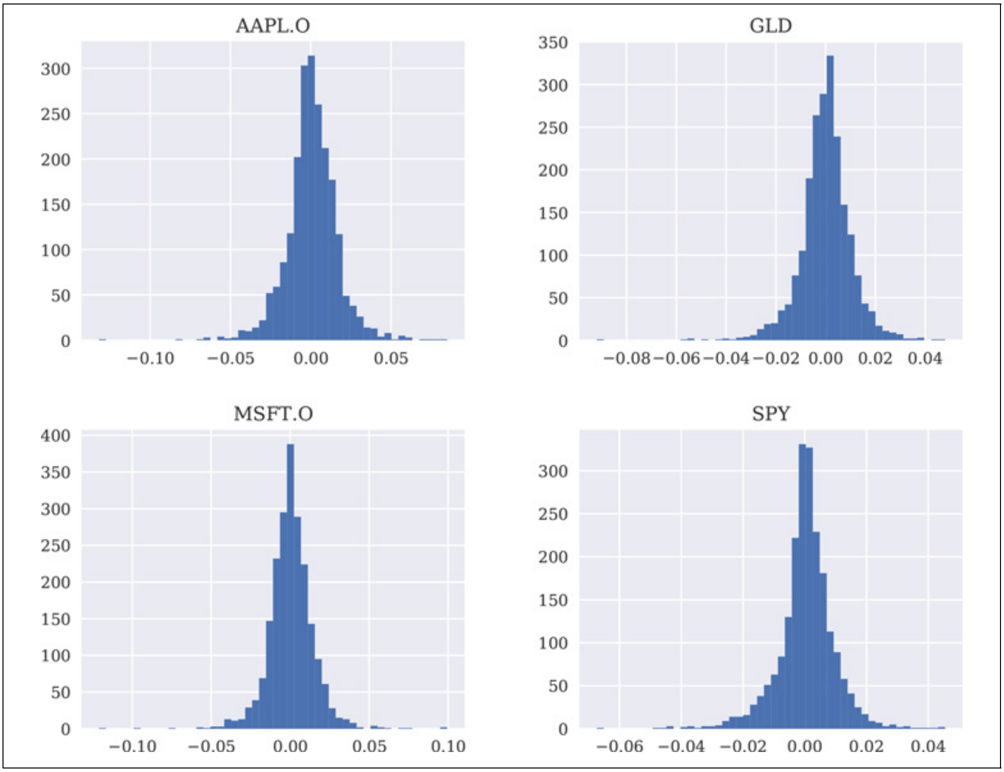


图 13-8　金融工具对数回报率直方图

接下来考虑时间序列数据集的不同统计量。峰度值似乎在所有 4 个数据集上都与正态分布的要求相去甚远：

```
In [36]: for sym in symbols:
             print('\nResults for symbol {}'.format(sym))
```

```
print(30 * '-')
log_data = np.array(log_returns[sym].dropna())
print_statistics(log_data) ❶
Results for symbol SPY
------------------------------
     statistic          value
------------------------------
          size     2137.00000
           min       -0.06734
           max        0.04545
          mean        0.00041
           std        0.00933
          skew       -0.52189
      kurtosis        4.52432

Results for symbol GLD
------------------------------
      statistic         value
------------------------------
           size    2137.00000
            min      -0.09191
            max       0.04795
           mean       0.00004
            std       0.01020
           skew      -0.59934
       kurtosis       5.68423

Results for symbol AAPL.O
------------------------------
      statistic         value
------------------------------
           size    2137.00000
           min       -0.13187
           max        0.08502
           mean       0.00084
           std        0.01591
           skew      -0.23510
       kurtosis       4.78964

Results for symbol MSFT.O
------------------------------
      statistic         value
------------------------------
           size    2137.00000
```

```
min       -0.12103
max        0.09941
mean       0.00054
std        0.01421
skew      -0.09117
kurtosis   7.29106
```

❶ 金融工具时间序列的统计量。

图 13-9 展示了 SPY ETF 的 QQ 图。很显然，样本的分位数值不在一条直线上，表明“非正态性”。在左侧和右侧分别有许多值远低于和远高于直线。换言之，这一时间序列信息出现了“大尾巴”（Fat tails）。大尾巴一词指的是（频率）分布中观察到的正负异常值远多于正态分布应有表现的情况。图 13-10 提供了 Microsoft 公司股票的数据，可以从中得出相同的结论，分布中也出现了明显的“大尾巴”现象：

```
In [37]: sm.qqplot(log_returns['SPY'].dropna(), line='s')
         plt.title('SPY')
         plt.xlabel('theoretical quantiles')
         plt.ylabel('sample quantiles');
In [38]: sm.qqplot(log_returns['MSFT.O'].dropna(), line='s')
         plt.title('MSFT.O')
         plt.xlabel('theoretical quantiles')
         plt.ylabel('sample quantiles');
```

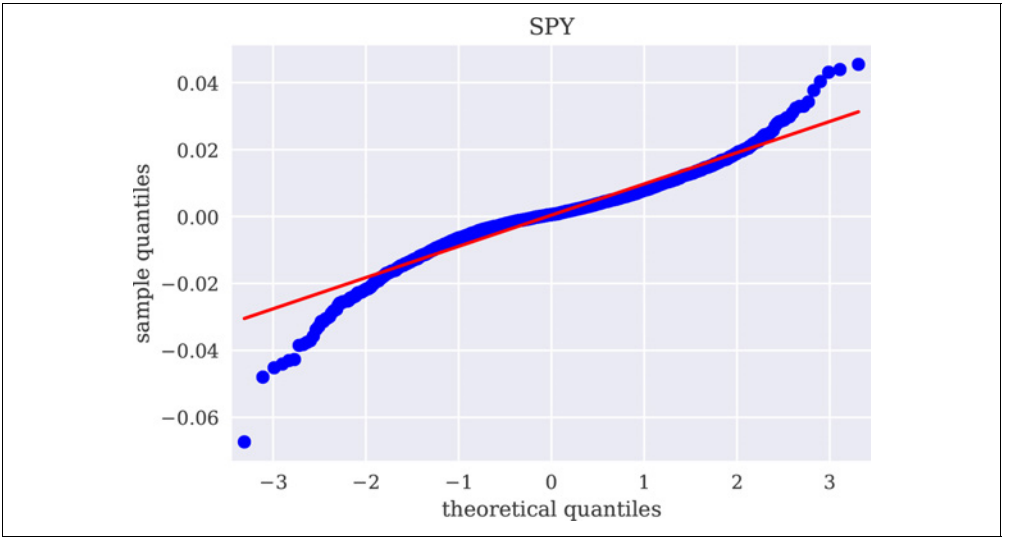


图 13-9　SPY 对数收益率分位数-分位数图

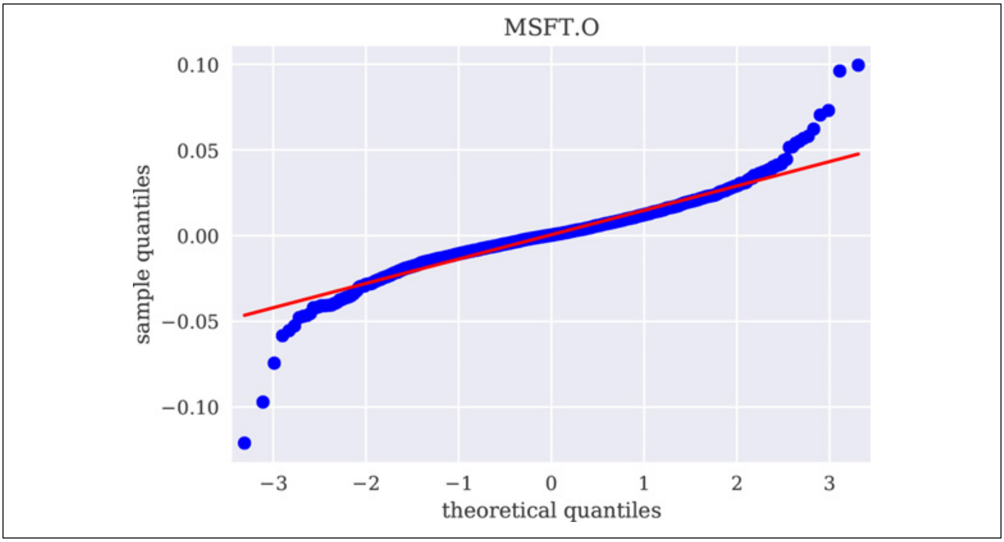


图 13-10　Microsoft 对数收益率分位数-分位数图

最后用上述结果进行统计正态性检验：

```
In [39]: for sym in symbols:
             print('\nResults for symbol {}'.format(sym))
             print(32 * '-')
             log_data = np.array(log_returns[sym].dropna())
             normality_tests(log_data) ❶

         Results for symbol SPY
         --------------------------------
         Skew of data set         -0.522
         Skew test p-value         0.000
         Kurt of data set          4.524
         Kurt test p-value         0.000
         Norm test p-value         0.000

         Results for symbol GLD
         --------------------------------
         Skew of data set         -0.599
         Skew test p-value         0.000
         Kurt of data set          5.684
         Kurt test p-value         0.000
         Norm test p-value         0.000

         Results for symbol AAPL.O
         --------------------------------
```

```
Skew of data set         -0.235
Skew test p-value         0.000
Kurt of data set          4.790
Kurt test p-value         0.000
Norm test p-value         0.000

Results for symbol MSFT.O
--------------------------------
Skew of data set         -0.091
Skew test p-value         0.085
Kurt of data set          7.291
Kurt test p-value         0.000
Norm test p-value         0.000
```

❶ 金融工具时间序列的正态性检验结果。

自始至终，不同测试的 p 值都为 0，强烈否决不同样板数据集呈正态分布的测试假设。这说明，股票市场收益率的正态假设——例如几何布朗运动模型中的假设——通常无法证明是正确的，可能需要使用可以产生“大尾巴”的更丰富模型（例如，跳跃扩散模型或者具备随机波动率的模型）。

13.2 投资组合优化

均值-方差投资组合理论（MPT）是金融理论的重要基础。因为这一理论的出现，所以 1990 年的诺贝尔经济学奖授予了其发明人 Harry Markowitz。尽管这种方法是 20 世纪 50 年代提出的[1]，但时至今日它仍然是一种金融学学生必须要学习并在实践中应用（往往有或大或小的改良）的理论。本节将阐述这种理论的基本原理。

由 Copeland、Wetson 和 Shastri（2005）合著的图书的第 5 章很好地介绍了与 MPT 相关联的正式主题。正如前面所指出的，正态分布收益率假设是该理论的基础：

> 只观察均值和方差，我们必然假定没有必要用其他统计量描述期末财富的分布。除非投资者有特殊的效用函数（二次效用函数），否则有必要假设收益率呈正态分布，正态分布用均值和方差就可以完整地描述。

13.2.1 数据

下面的分析和示例使用了和前面相同的金融工具。MPT 的基本思路是通过分散投资实现投资组合风险最小化，或者在指定风险水平下的组合收益最大化。通过合理地组合足够大量的资产并让资产多样化，可以实现预期的结果。为了表明基本思路并说明典

1 参见 Markowitz，Harry（1952）：“*Portfolio Selection*”，*Journal of Finance*，第 7 卷，77-91 页。——原注

型的效果，4 种资产应该足够了。图 13-11 展示了金融工具对数回报率的频率分布：

```
In [40]: symbols = ['AAPL.O', 'MSFT.O', 'SPY', 'GLD'] ❶

In [41]: noa = len(symbols) ❷

In [42]: data = raw[symbols]

In [43]: rets = np.log(data / data.shift(1))

In [44]: rets.hist(bins=40, figsize=(10, 8));
```

❶ 构成投资组合的 4 种金融工具。

❷ 定义的金融工具数量。

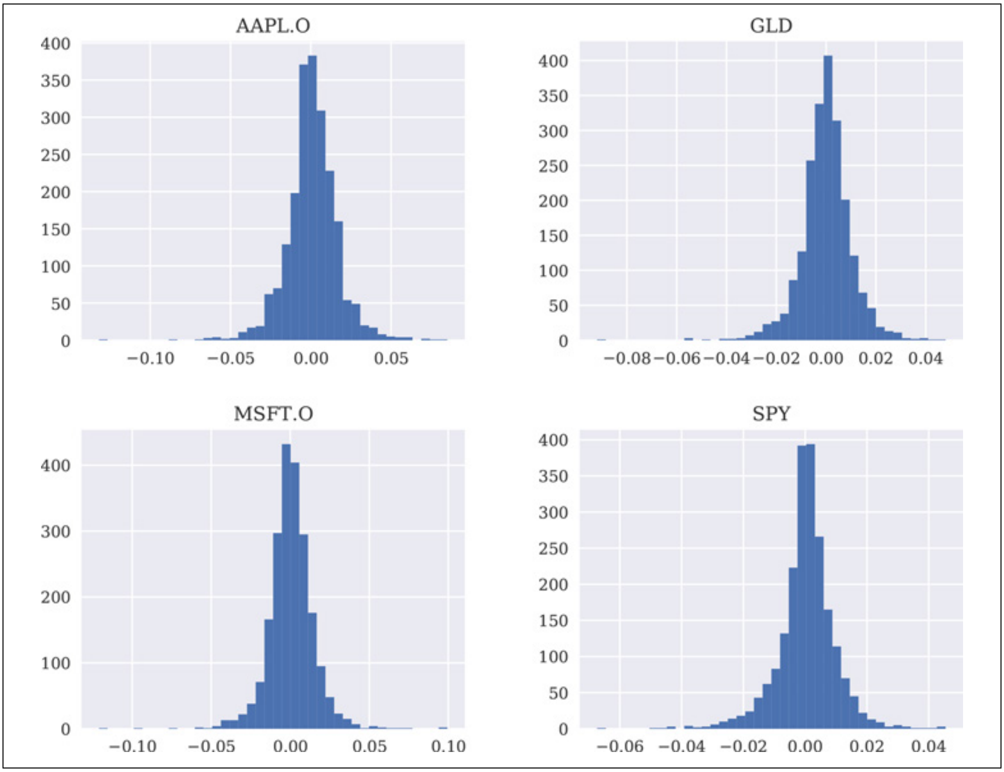


图 13-11　金融工具对数回报率直方图

投资资产的协方差矩阵是整个投资组合选择过程的核心部分。pandas 有一个内建方法，可以生成协方差矩阵，该矩阵应用相同的比例调整因子：

```
In [45]: rets.mean() * 252 ❶
Out[45]: AAPL.O    0.212359
```

```
          MSFT.O    0.136648
           SPY      0.102928
           GLD      0.009141
           dtype: float64

In [46]: rets.cov() * 252 ❷
Out[46]:           AAPL.O    MSFT.O       SPY       GLD
         AAPL.O  0.063773  0.023427  0.021039  0.001513
         MSFT.O  0.023427  0.050917  0.022244 -0.000347
         SPY     0.021039  0.022244  0.021939  0.000062
         GLD     0.001513 -0.000347  0.000062  0.026209
```

❶ 年化平均收益率。

❷ 年化协方差矩阵。

13.2.2　基本理论

下面，我们假定投资者不允许在某种金融工具上建立空头头寸。只允许多头头寸意味着投资者的财富将在可用资产中分配，所有头寸均为多头（正）头寸，且头寸的总和为 100%。例如，可以在每种工具中投入相同的资金量（每种 25%）。下面的代码生成 4 个 0 和 1 之间的随机数，然后对这些数值进行规范化，使所有值的总和为 1：

```
In [47]: weights = np.random.random(noa) ❶
         weights /= np.sum(weights) ❷

In [48]: weights
Out[48]: array([0.07650728, 0.06021919, 0.63364218, 0.22963135])

In [49]: weights.sum()
Out[49]: 1.0
```

❶ 随机投资组合权重。

❷ 将其规范化为 1 或者 100%。

现在可以检查资产权重总和确实为 1；也就是说 $\Sigma_I w_i = 1$，其中 I 是资产的数量，$w_i > 0$ 是资产的权重。公式 13-1 提供在给定单一证券权重情况下的预期投资组合收益。这个预期投资收益公式假定，历史平均表现是未来（预期）表现的最佳预测因素。在公式 13-1 中，r_i 是状态相关未来收益（由假定为正态分布的收益值组成的向量），而 μ_i 是证券 i 的预期收益。最后，$\mathbf{w}^T$ 是权重向量的转置，μ 是预期证券收益的向量。

公式 13-1　预期收益一般公式

$$\mu_p = E\left(\sum_I w_i r_i\right)$$

$$= \sum_{I} w_i E(r_i)$$

$$= \sum_{I} w_i \mu_i$$

$$= \boldsymbol{w}^{\mathrm{T}} \mu$$

将其转换为 Python 代码，得到如下的代码行，已包含年化收益值：

```
In [50]: np.sum(rets.mean() * weights) * 252 ❶
Out[50]: 0.09179459482057793
```

❶ 给定投资组合权重下的年化收益率。

在 MPT 中选择的第二个对象是预期投资组合方差。两种证券间的协方差定义为 $\sigma_{ij} = \sigma_{ji} = E(r_i - \mu_i)(r_j - \mu_j)$。证券的方差是其与自身的协方差：$\sigma_i^2 = E((r_i - \mu_i)^2)$。公式 13-2 提供了一个证券投资组合的协方差矩阵（假定每种证券的权重均为 1）。

公式 13-2　投资组合协方差矩阵

$$\sum = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1I} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2I} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{I1} & \sigma_{I2} & \cdots & \sigma_I^2 \end{bmatrix}$$

利用投资组合协方差矩阵后，公式 13-3 提供了预期投资组合方差的公式。

公式 13-3　预期投资组合方差一般公式

$$\sigma_p^2 = E((r - \mu)^2)$$

$$= \sum_{i \in I} \sum_{j \in I} w_i w_j \sigma_{ij}$$

$$= \boldsymbol{w}^{\mathrm{T}} \sum w$$

在 Python 中，以上公式可以利用 NumPy 的向量化功能归结为一行代码。`np.dot()`函数给出两个向量/矩阵的点乘。`T` 属性或者 `transpose()`方法给出向量或者矩阵的转置。在投资组合方差给定的情况下，（预期）投资组合标准差（波动率）只需要计算一次平方根即可得到（$\sigma_p = \sqrt{\sigma_p^2}$）：

```
In [51]: np.dot(weights.T, np.dot(rets.cov() * 252, weights)) ❶
Out[51]: 0.014763288666485574

In [52]: math.sqrt(np.dot(weights.T, np.dot(rets.cov() * 252, weights))) ❷
Out[52]: 0.12150427427249452
```

❶ 给定权重下的年化投资组合方差。

❷ 给定权重下的年化投资组合波动率。

Python 与向量化

MPT 示例再次展示了 Python 将数学概念（如投资组合收益或者投资组合方差）转换为可执行向量化代码的效率（第 1 章的论点之一）。

大体上完成了均值-方差投资组合选择所需的工具集。投资者最感兴趣的是给定证券组合的风险-收益均衡性及其统计学特性。为此，我们进行一次蒙特卡洛模拟（参见第 12 章），以生成较大规模的随机投资组合权重向量。对于每一种模拟的分配，我们记录得出的预期投资组合收益和方差。为了简化代码，定义两个函数 `port_ret()` 和 `port_vol()`：

```
In [53]: def port_ret(weights):
             return np.sum(rets.mean() * weights) * 252

In [54]: def port_vol(weights):
             return np.sqrt(np.dot(weights.T, np.dot(rets.cov() *
252, weights)))

In [55]: prets = []
         pvols = []
         for p in range (2500): ❶
             weights = np.random.random(noa) ❶
             weights /= np.sum(weights) ❶
             prets.append(port_ret(weights)) ❷
             pvols.append(port_vol(weights)) ❷
         prets = np.array(prets)
         pvols = np.array(pvols)
```

❶ 投资组合权重的蒙特卡洛模拟。

❷ 将结果统计量收集在列表对象中。

图 13-12 展示了蒙特卡洛模拟的结果。除此之外，还提供了定义为 $SR \equiv \frac{\mu_p - r_f}{\sigma_p}$ 的夏普指数（即预期投资组合超额收益），也就是投资收益率超过无风险短期利率 r_f 的部分除以预期投资组合标准差。简单起见，假定 $r_f \equiv 0$：

```
In [56]: plt.figure(figsize=(10, 6))
         plt.scatter(pvols, prets, c=prets / pvols,
                     marker='o', cmap='coolwarm')
         plt.xlabel('expected volatility')
         plt.ylabel('expected return')
         plt.colorbar(label='Sharpe ratio');
```

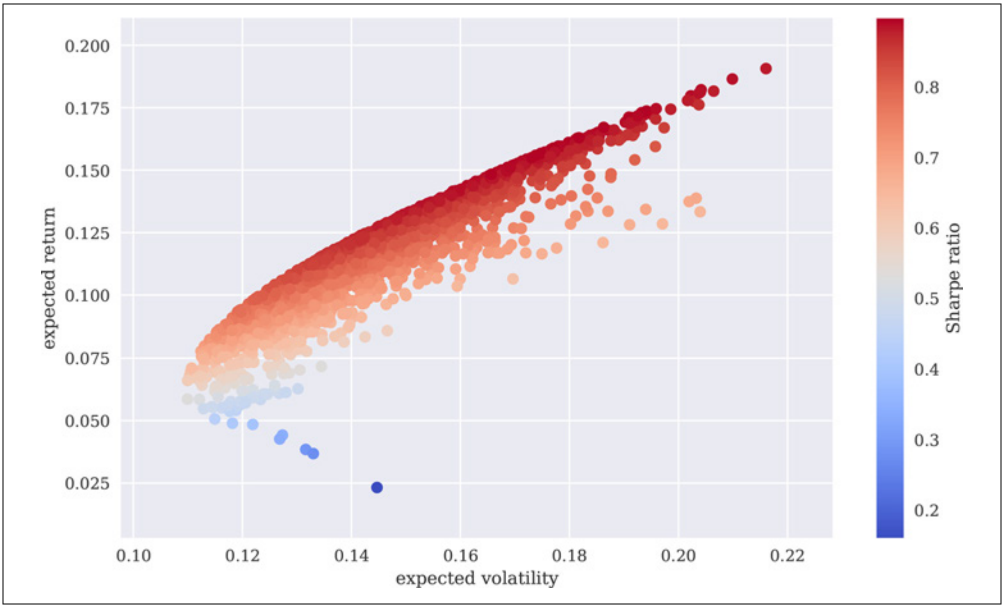


图 13-12　随机投资组合权重的预期收益和波动率

通过图 13-12 可以明显地看出，从均值和方差考量，并不是所有权重分布都表现良好。例如，对于固定风险水平（如 15%），多种组合都表现出不同的收益。作为投资者，人们通常对固定风险水平的最大收益率或者固定收益率预期下的最小风险感兴趣。这两种投资组合组成所谓的有效边界，这是本节后面想要得到的结果。

13.2.3　最优投资组合

最小化函数很通用，它考虑了参数的（不）等式约束和参数的范围。

我们从夏普指数的最大化开始。从形式上说，最小化夏普指数的负值可以得到最大价值和最优投资组合构成。约束是所有参数（权重）的总和为 1。用 minimize 函数的约定表达后面会讲到（参见 minimize 函数文档）[1]。我们还将参数值（权重）限制在 0 和 1 之间，这些值以多个元组组成的一个元组形式提供给最小化函数。

优化函数调用中忽略的唯一输入是起始参数列表（对权重的初始猜测）。我们简单地使用平均分布：

```
In [57]: import scipy.optimize as sco

In [58]: def min_func_sharpe(weights): ❶
             return -port_ret(weights) / port_vol(weights) ❶
```

1　`np.sum(x)-1` 的替代写法是 `np.sum(x) == 1`，这种写法考虑到 Python 的布尔值 True 等于 1，False 等于 0。——原注

```
In [59]: cons = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1}) ❷

In [60]: bnds = tuple((0, 1) for x in range(noa)) ❸

In [61]: eweights = np.array(noa * [1. / noa,]) ❹
         eweights ❹
Out[61]: array([0.25, 0.25, 0.25, 0.25])

In [62]: min_func_sharpe(eweights)
Out[62]: -0.8436203363155397
```

❶ 需要最小化的函数。

❷ 等式约束。

❸ 参数范围。

❹ 等权重向量。

调用该函数不仅返回最优化参数值，还返回其他很多信息。我们将结果存储在 opts 对象中。我们的主要兴趣点在于获得最优投资组合的构成。为此，提供自己感兴趣的键值（下列代码中是 x）来访问结果对象：

```
In [63]: %%time
         opts = sco.minimize(min_func_sharpe, eweights,
                             method='SLSQP', bounds=bnds,
                             constraints=cons) ❶
         CPU times: user 67.6 ms, sys: 1.94 ms, total: 69.6 ms
         Wall time: 75.2 ms

In [64]: opts ❷
Out[64]:      fun: -0.8976673894052725
         jac: array([ 8.96826386e-05, 8.30739737e-05, -2.45958567e-04,
          1.92895532e-05])
          message: 'Optimization terminated successfully.'
             nfev: 36
              nit: 6
             njev: 6
           status: 0
          success: True
                x: array([0.51191354, 0.19126414, 0.25454109, 0.04228123])

In [65]: opts['x'].round(3) ❸
Out[65]: array([0.512, 0.191, 0.255, 0.042])

In [66]: port_ret(opts['x']).round(3) ❹
```

```
Out[66]: 0.161

In [67]: port_vol(opts['x']).round(3) ❺
Out[67]: 0.18

In [68]: port_ret(opts['x']) / port_vol(opts['x']) ❻
Out[68]: 0.8976673894052725
```

❶ 优化（即函数 `min_func_sharpe()`的最小化）。

❷ 优化结果。

❸ 最优投资组合权重。

❹ 得出的投资组合回报率。

❺ 得出的投资组合波动率。

❻ 最大夏普指数。

接下来，最小化投资组合的方差，这与波动率的最小化相同：

```
In [69]: optv = sco.minimize(port_vol, eweights,
                             method='SLSQP', bounds=bnds,
                             constraints=cons) ❶

In [70]: optv
Out[70]:     fun: 0.1094215526341138
             jac: array([0.11098004, 0.10948556, 0.10939826, 0.10944918])
         message: 'Optimization terminated successfully.'
            nfev: 54
             nit: 9
            njev: 9
          status: 0
         success: True
        x: array([1.62630326e-18, 1.06170720e-03, 5.43263079e-01,
         4.55675214e-01])

In [71]: optv['x'].round(3)
Out[71]: array([0.   , 0.001, 0.543, 0.456])

In [72]: port_vol(optv['x']).round(3)
Out[72]: 0.109

In [73]: port_ret(optv['x']).round(3)
Out[73]: 0.06

In [74]: port_ret(optv['x']) / port_vol(optv['x'])
```

```
Out[74]: 0.5504173653075624
```

❶ 投资组合波动率的最小化。

这次，投资组合仅由 3 种金融工具构成。这种组合可以得到所谓的绝对最小方差投资组合。

13.2.4 有效边界

所有最优化投资组合——即目标收益率水平下波动率最小的所有投资组合（或者给定风险水平下收益率最大的所有投资组合）——的求取都和上述最优化过程类似。唯一的区别是我们必须循环使用多种起始条件。

我们所采取的方法是固定目标回报率水平，得出每种水平下导致最小波动率值的组合权重。对于优化而言，这产生两种条件：一是目标收益率水平 `tret`，另一个是投资组合权重的总和。每个参数的边界值保持不变。在不同目标收益率水平（`tret`）中循环时，最小的一个条件会变化。这就是在每次循环中更新条件字典对象的原因：

```
In [75]: cons = ({'type': 'eq', 'fun': lambda x: port_ret(x) - tret},
                 {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}) ❶

In [76]: bnds = tuple((0, 1) for x in weights)

In [77]: %%time
         trets = np.linspace(0.05, 0.2, 50)
         tvols = []
         for tret in trets:
             res = sco.minimize(port_vol, eweights, method='SLSQP',
                                bounds=bnds, constraints=cons) ❷
             tvols.append(res['fun'])
         tvols = np.array(tvols)
         CPU times: user 2.6 s, sys: 13.1 ms, total: 2.61 s
         Wall time: 2.66 s
```

❶ 有效边界的两个绑定约束。

❷ 不同目标回报率的组合波动率优化。

图 13-13 展示了最优化的结果。交叉表示给定某个目标收益率的最优投资组合；小点和以前一样表示随机组合。此外，该图还出现了两个较大的星号：一个表示最小波动率/方差投资组合（最左侧的组合），另一个表示具有最大夏普指数的投资组合：

```
In [78]: plt.figure(figsize=(10, 6))
         plt.scatter(pvols, prets, c=prets / pvols,
                     marker='.', alpha=0.8, cmap='coolwarm')
         plt.plot(tvols, trets, 'b', lw=4.0)
```

```
plt.plot(port_vol(opts['x']), port_ret(opts['x']),
         'y*', markersize=15.0)
plt.plot(port_vol(optv['x']), port_ret(optv['x']),
         'r*', markersize=15.0)
plt.xlabel('expected volatility')
plt.ylabel('expected return')
plt.colorbar(label='Sharpe ratio')
```

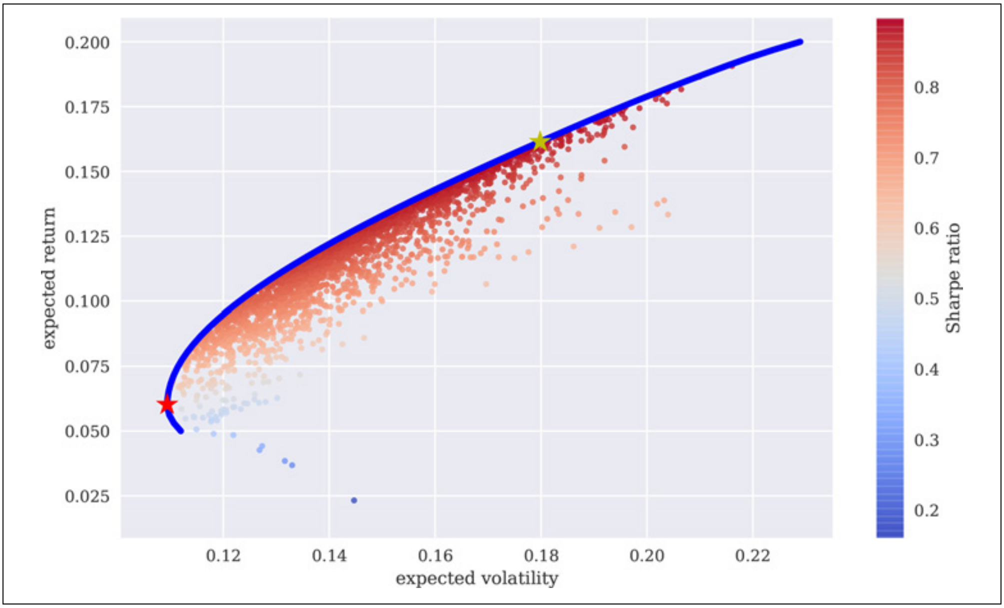


图 13-13　给定收益率水平的最小风险投资组合（有效边界）

有效边界由所有收益率高于绝对最小方差投资组合的最优投资组合构成。这些投资组合在给定某一风险水平的预期收益率上优于其他所有投资组合。

13.2.5　资本市场线

除了股票等高风险证券或者商品（如黄金）之外，通常有一种普遍的无风险投资机会：现金或者现金账户。在理想化的世界中，保存在大银行现金账户中的资金可以认为是无风险的（例如，通过公共存款保险计划）。这种无风险投资的缺点是通常只能得到很少的收益，有时接近于 0。

然而，考虑这些无风险资产可以显著加强投资者的有效投资机会。基本思路是，投资者首先确定高风险资产的一个有效组合，然后在组合中加入无风险资产。通过调整投资于无风险资产中的财富比例，有可能实现任何风险-收益均衡性，这些配置位于（风险-收益空间中）无风险资产和有效投资组合之间的直线上。

（在许多选项中）使用哪一个有效投资组合来进行最优化风格的投资呢？选择有效边界切线恰好通过无风险投资组合风险-收益点的投资组合。例如，考虑无风险利率 $r_f = 0.01$ 的情况。我们寻找有效边界上切线穿过风险-收益空间上的点（σ_f，r_f）=（0，0.01）的投资组合。

为了进行下面的计算，我们需要一个函数逼近和有效边界的一阶导数。3 次样条插值提供了这种可导函数逼近（参见第 11 章）。对于样条插值，我们只使用有效边界中的投资组合。通过这条数值化路径，最终可以为有效边界定义一个连续可导函数 `f(x)`和对应的一阶导数函数 `df(x)`：

```
In [79]: import scipy.interpolate as sci

In [80]: ind = np.argmin(tvols) ❶
         evols = tvols[ind:] ❷
         erets = trets[ind:] ❷

In [81]: tck = sci.splrep(evols, erets) ❸

In [82]: def f(x):
             ''' Efficient frontier function (splines approximation). '''
             return sci.splev(x, tck, der=0)
         def df(x):
             ''' First derivative of efficient frontier function. '''
             return sci.splev(x, tck, der=1)
```

❶ 最小波动率投资组合的指数头寸。

❷ 相关的组合波动率和回报率值。

❸ 这些值上的 3 次样条插值。

现在要得出的是线性函数 $t(x) = a+b \cdot x$，这代表着穿过风险-收益空间中无风险资产并与有效边界相切的一条直线。公式 13-4 描述了函数 $t(x)$必须满足的 3 个条件。

公式 13-4　资本市场线的数学条件

$$t(x) = a + b \cdot x$$

$$t(0) = r_f \quad \Leftrightarrow \quad a = r_f$$

$$t(x) = f(x) \quad \Leftrightarrow \quad a + b \cdot x = f(x)$$

$$t'(x) = f'(x) \quad \Leftrightarrow \quad b = f'(x)$$

由于我们没有有效边界的封闭公式或者它的一阶导数，所以必须以数值化方式求得公式 13-4 中方程组的解。为此，我们定义一个 Python 函数，返回给定参数集 `p = (a, b, x)` 下的全部 3 个方程式的值。

scipy.optimize 中的 sco.fsolve()函数能够解这类方程组。我们提供初始参数，以及 equations()函数。注意，优化的成败可能取决于初始参数，因此必须小心选择这些参数——通常需要结合合理的猜测，并反复尝试：

```
In [83]: def equations(p, rf=0.01):
             eq1 = rf - p[0] ❶
             eq2 = rf + p[1] * p[2] - f(p[2]) ❶
             eq3 = p[1] - df(p[2]) ❶
             return eq1, eq2, eq3

In [84]: opt = sco.fsolve(equations, [0.01, 0.5, 0.15]) ❷

In [85]: opt ❸
Out[85]: array([0.01      , 0.84470952, 0.19525391])

In [86]: np.round(equations(opt), 6) ❹
Out[86]: array([ 0., 0., -0.])
```

❶ 描述资本市场线（CML）的方程式。

❷ 按照给定初始值解方程。

❸ 最优参数值。

❹ 方程式值都为 0。

图 13-14 以图形方式提供了结果：星号代表有效边界中切线穿过无风险资产点（0, $r_f = 0.01$）的最优化投资组合。

```
In [87]: plt.figure(figsize=(10, 6))
         plt.scatter(pvols, prets, c=(prets - 0.01) / pvols,
                     marker='.', cmap='coolwarm')
         plt.plot(evols, erets, 'b', lw=4.0)
         cx = np.linspace(0.0, 0.3)
         plt.plot(cx, opt[0] + opt[1] * cx, 'r', lw=1.5)
         plt.plot(opt[2], f(opt[2]), 'y*', markersize=15.0)
         plt.grid(True)
         plt.axhline(0, color='k', ls='--', lw=2.0)
         plt.axvline(0, color='k', ls='--', lw=2.0)
         plt.xlabel('expected volatility')
         plt.ylabel('expected return')
         plt.colorbar(label='Sharpe ratio')
```

最优投资组合的权重如下，该组合只包含 4 种资产中的 3 种：

```
In [88]: cons = ({'type': 'eq', 'fun': lambda x: port_ret(x) - f(opt[2])},
                 {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}) ❶
         res = sco.minimize(port_vol, eweights, method='SLSQP',
```

```
                              bounds=bnds, constraints=cons)

In [89]: res['x'].round(3) ❷
Out[89]: array([0.59 , 0.221, 0.189, 0.  ])

In [90]: port_ret(res['x'])
Out[90]: 0.1749328414905194

In [91]: port_vol(res['x'])
Out[91]: 0.19525371793918325

In [92]: port_ret(res['x']) / port_vol(res['x'])
Out[92]: 0.8959257899765407
```

❶ 相切投资组合（图 13-14 中的金色星号）的绑定约束。

❷ 这个特殊投资组合的权重。

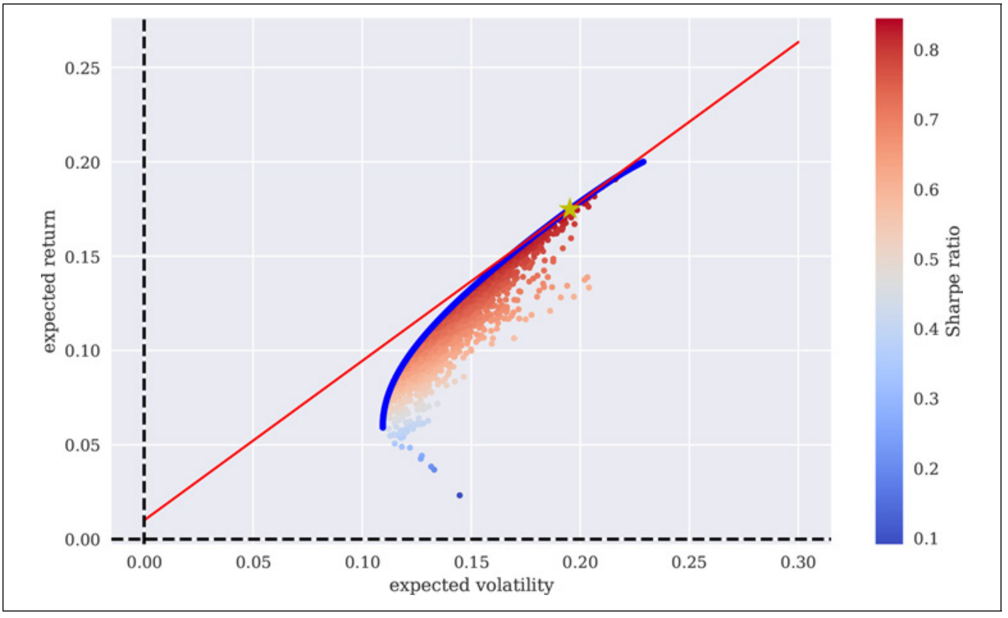


图 13-14　无风险利率为 1%时的资本市场线和相切的投资组合（星号）

13.3　贝叶斯统计

近年来，贝叶斯统计已经成为实证金融学的基石。本章无法覆盖该领域的所有概念。因此，在必要时应该参考 Geweke（2005）的教科书来了解一般的入门知识，利益驱动者应该参考 Rachev（2008）的教科书。

13.3.1 贝叶斯公式

贝叶斯公式在金融学中常见的解读是历时解读。这主要说明，随着时间的推移，我们会了解感兴趣的变量或者参数的新信息，例如时间序列的平均收益率。公式 13-5 正式描述了这种理论。

公式 13-5 贝叶斯公式

$$p(H \mid D) = \frac{p(H) \cdot p(D \mid H)}{p(D)}$$

式中 H 表示某个事件（假设），D 代表实验或者真实世界可能提供的数据[1]。在这些定义的基础上，我们得到：

- $p(H)$称作先验概率；
- $p(D)$是任何假设下数据的概率，称作标准化常数；
- $p(D|H)$是假设 H 下数据的似然度（即概率）；
- $p(H|D)$是后验概率，即我们看到数据之后得出的概率。

考虑一个简单的例子。有两个盒子 B_1 和 B_2，B_1 中有 20 个黑球和 70 个红球，而 B_2 中有 40 个黑球和 50 个红球。随机从两个盒子中取出一个球，假定这个球是黑色的。“H_1：球来自 B_1”和“H_2：球来自 B_2”这两个假设的概率分别是多少？

在随机取出这个球之前，两个假设的可能性是相同的。但当球明显是黑色，我们必须根据贝叶斯公式更新两个假设的概率，考虑假设 H_1。

- 先验概率：$p(H_1) = 1/2$
- 标准化常数：$p(D) = \frac{1}{2} \times \frac{1}{3} + \frac{1}{2} \times \frac{2}{3} = \frac{1}{2}$
- 似然度：$p(D|H_1) = \frac{1}{3}$

更新后的 H_1 概率 $p(H_1|D) = \frac{\frac{1}{2} \times \frac{1}{3}}{\frac{1}{2}} = \frac{1}{3}$

这一结果也有直观的意义。从 B_2 取出一个黑球的概率两倍于同一件事情发生在 B_1 上的概率。因此，取出一个黑球之后，假设 H_2 的更新后概率 $p(H_2|D)=2/3$，也是假设 H_1 更新后概率的 2 倍。

1 基于 Python 的贝叶斯统计学基础概念介绍参见 Downey（2013）。——原注

13.3.2　贝叶斯回归

Python 生态系统利用 PyMC3 提供了一个全面的软件包，从技术上实现了贝叶斯统计和概率规划。

考虑如下基于直线周围噪声数据的例子。[1]首先，在数据集上实施一次线性普通最小二乘回归（参见第 11 章），结果如图 13-15 所示：

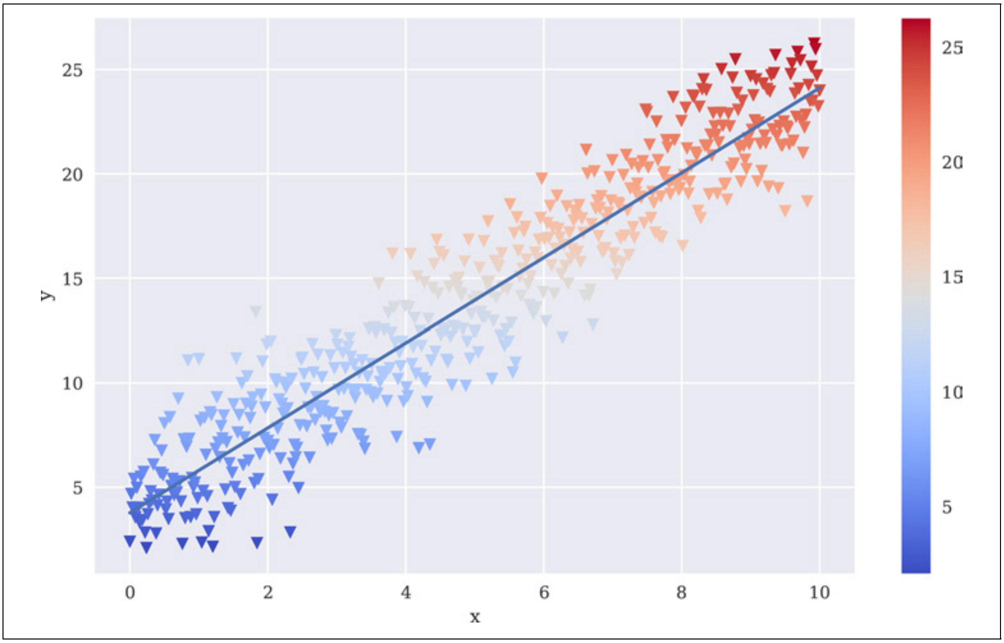


图 13-15　样本数据点和回归线

```
In [1]: import numpy as np
        import pandas as pd
        import datetime as dt
        from pylab import mpl, plt

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        np.random.seed(1000)
        %matplotlib inline

In [3]: x = np.linspace(0, 10, 500)
        y = 4 + 2 * x + np.random.standard_normal(len(x)) * 2
```

1　这个例子最初由 Thomas Wiceki 提供，他是 PyMC3 软件包的主要作者之一。——原注

```
In [4]: reg = np.polyfit(x, y, 1)

In [5]: reg
Out[5]: array([2.03384161, 3.77649234])

In [6]: plt.figure(figsize=(10, 6))
        plt.scatter(x, y, c=y, marker='v', cmap='coolwarm')
        plt.plot(x, reg[1] + reg[0] * x, lw=2.0)
        plt.colorbar()
        plt.xlabel('x')
        plt.ylabel('y')
```

OLS 回归方法的结果是回归线上两个参数（截距和斜率）的固定值。注意，最高阶单项式因子（本例中是回归线的斜率）在指数水平 0 处，截距在指数水平 1 处。原始参数 2 和 4 没能完全恢复，这是数据中包含的噪声所致。

贝叶斯回归利用了 PyMC3 软件包。这里假定参数遵循某种分布。例如，考虑回归线方程 $\hat{y}(x)=\alpha+\beta\cdot x$ 。假定现在有以下先验概率：

- α 呈正态分布，均值为 0，标准差为 20；
- β 呈正态分布，均值为 0，标准差为 10。

至于似然度，假定一个均值为 $\hat{y}(x)$ 的正态分布和一个标准差为 0～10 之间的均匀分布。

贝叶斯回归的一个要素是（马尔科夫链）蒙特卡洛（MCMC）采样[1]。原则上，这和前一个例子中多次从盒子取出球相同——只是采用更系统性、更自动化的方式。

技术性采样有 3 个不同的函数可以调用：

- `find_MAP()`通过求取局部最大后验点来寻找采样算法的起点；
- `NUTS()`为假定先验概率的 MCMC 采样实现所谓的“高效双平均无回转采样器”（NUTS）算法；
- `sample()`以给定起始值（来自 `find_MAP()`）和最优化步长（来自 NUTS 算法）提取一定数量的样本。

上列函数都包装到一个 `PyMC3 Model` 对象中，并在 `with` 语句中执行：

```
In [8]: import pymc3 as pm

In [9]: %%time
```

1　参见维基百科的马尔科夫链。例如，本书自始至终使用在第 12 章中详细分析的蒙特卡洛算法来生成所谓的马尔科夫链，因为下一步骤/值取决于过程的当前状态，而非其他历史状态或者值。——原注

```
with pm.Model() as model:
    # model
    alpha = pm.Normal('alpha', mu=0, sd=20) ❶
    beta = pm.Normal('beta', mu=0, sd=10) ❶
    sigma = pm.Uniform('sigma', lower=0, upper=10) ❶
    y_est = alpha + beta * x ❷
    likelihood = pm.Normal('y', mu=y_est, sd=sigma,
                           observed=y) ❸

    # inference
    start = pm.find_MAP() ❹
    step = pm.NUTS() ❺
    trace = pm.sample(100, tune=1000, start=start,
                      progressbar=True, verbose=False) ❻
logp = -1,067.8, ||grad|| = 60.354: 100%|█████| 28/28 [00:00<00:00,
 474.70it/s]
Only 100 samples in chain.
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sigma, beta, alpha]
Sampling 2 chains: 100%|█████| 2200/2200 [00:03<00:00,
 690.96draws/s]

CPU times: user 6.2 s, sys: 1.72 s, total: 7.92 s
Wall time: 1min 28s

In [10]: pm.summary(trace) ❼
Out[10]:
             mean        sd  mc_error  hpd_2.5  hpd_97.5       n_eff      Rhat
    alpha 3.764027 0.174796 0.013177 3.431739  4.070091 152.446951 0.996281
    beta  2.036318 0.030519 0.002230 1.986874  2.094008 106.505590 0.999155
    sigma 2.010398 0.058663 0.004517 1.904395  2.138187 188.643293 0.998547

In [11]: trace[0] ❽
Out[11]: {'alpha': 3.9303300798212444,
          'beta': 2.0020264758995463,
          'sigma_interval__': -1.3519315719461853,
          'sigma': 2.0555476283253156}
```

❶ 定义先验概率。

❷ 指定线性回归。

❸ 定义似然度。

❹ 通过优化找出起始值。

❺ 实例化 MCMC 算法。

❻ 用 NUTS 取得后验样本。

❼ 展示采样的统计摘要。

❽ 从第一个样本估算。

3 个估算值都相当接近原始值（4，2，2）。不过，整个过程会得到许多估算值。最好用轨迹图（如图 13-16 所示）描述它们。轨迹图展示了不同参数得到的后验分布以及每个样本的单独估算值。后验分布帮助我们直观地了解了估算值中的不确定性：

```
In [12]: pm.traceplot(trace, lines={'alpha': 4, 'beta': 2, 'sigma': 2});
```

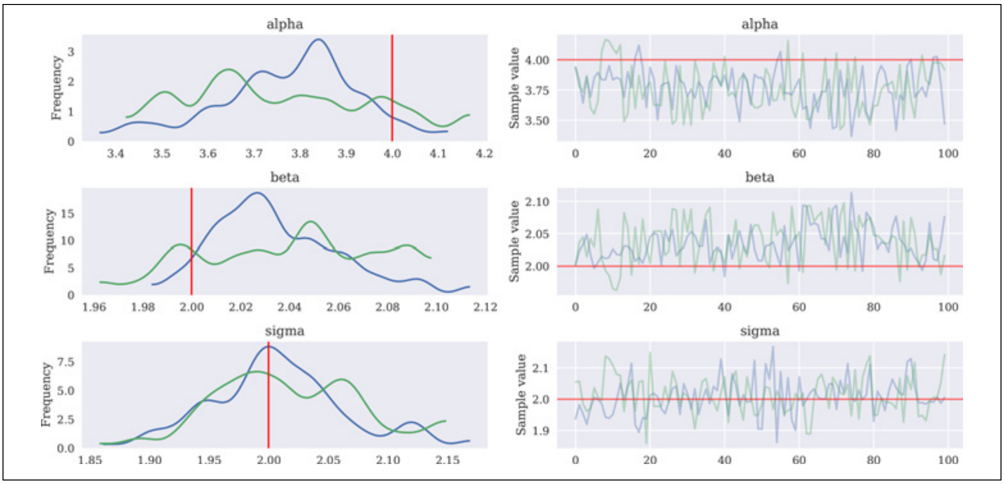


图 13-16　后验分布及轨迹图

仅从回归中取得 `alpha` 和 `beta` 值就可以绘制所有的结果回归线（见图 13-17）：

```
In [13]: plt.figure(figsize=(10, 6))
         plt.scatter(x, y, c=y, marker='v', cmap='coolwarm')
         plt.colorbar()
         plt.xlabel('x')
         plt.ylabel('y')
         for i in range(len(trace)):
             plt.plot(x, trace['alpha'][i] + trace['beta'][i] * x) ❶
```

❶ 绘制单条回归线。

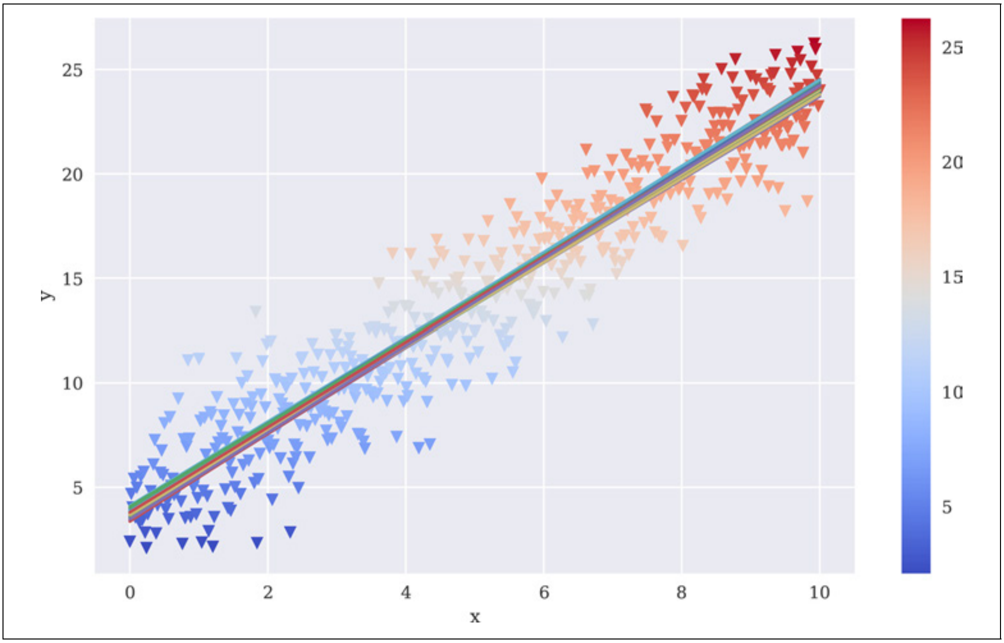


图 13-17　基于不同估算值的回归线

13.3.3　两种金融工具

用虚拟数据介绍了 PyMC3 贝叶斯回归之后，使用真实的金融数据很简单。示例使用了两种交易型开放式基金（GLD 和 GDX）的金融时间序列数据（见图 13-18）：

```
In [14]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                           index_col=0, parse_dates=True)

In [15]: data = raw[['GDX', 'GLD']].dropna()

In [16]: data = data / data.iloc[0] ❶

In [17]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2138 entries, 2010-01-04 to 2018-06-29
         Data columns (total 2 columns):
         GDX    2138 non-null float64
         GLD    2138 non-null float64
         dtypes: float64(2)
         memory usage: 50.1 KB

In [18]: data.ix[-1] / data.ix[0] - 1 ❷
Out[18]: GDX   -0.532383
```

```
         GLD    0.080601
         dtype: float64

In [19]: data.corr() ❸
Out[19]:          GDX      GLD
         GDX  1.00000  0.71539
         GLD  0.71539  1.00000

In [20]: data.plot(figsize=(10, 6));
```

❶ 将数据规范化，起始值 1。

❷ 计算相对绩效。

❸ 计算两种金融工具之间的相关度。

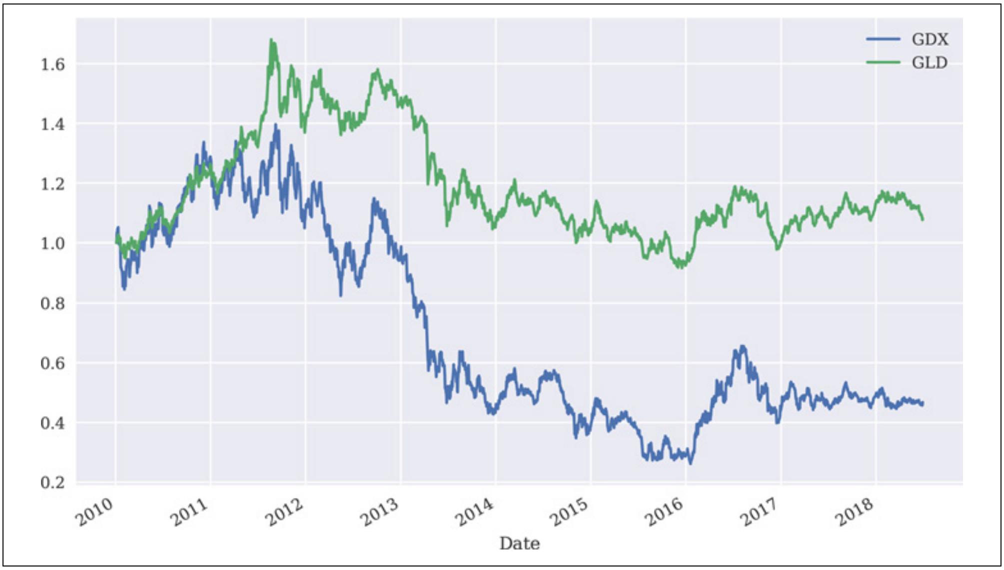


图 13-18　GLD 和 GDX 一段时间后的规范化价格

在下面的例子中，单个数据点的日期用散点图来可视化。为此，将 DataFrame 中的 DatetimeIndex 对象转换成 matplotlib 日期。图 13-19 展示了时间序列数据的散点图，并将 GLD 价值与 GDX 价值进行对照，然后以不同颜色显示每对数据的日期：[1]

```
In [21]: data.index[:3]
Out[21]: DatetimeIndex(['2010-01-04', '2010-01-05', '2010-01-06'],
          dtype='datetime64[ns]', name='Date', freq=None)
```

1　注意，这里的可视化使用规范化数据的日期，而在实际应用中使用收益率数据的日期可能更好。——原注

```
In [22]: mpl_dates = mpl.dates.date2num(data.index.to_pydatetime()) ❶
         mpl_dates[:3]
Out[22]: array([733776., 733777., 733778.])

In [23]: plt.figure(figsize=(10, 6))
         plt.scatter(data['GDX'], data['GLD'], c=mpl_dates,
                     marker='o', cmap='coolwarm')
         plt.xlabel('GDX')
         plt.ylabel('GLD')
         plt.colorbar(ticks=mpl.dates.DayLocator(interval=250),
                      format=mpl.dates.DateFormatter('%d %b %y')); ❷
```

❶ 将 `DatetimeIndex` 对象转换为 `matplotlib` 日期。

❷ 自定义日期的色卡。

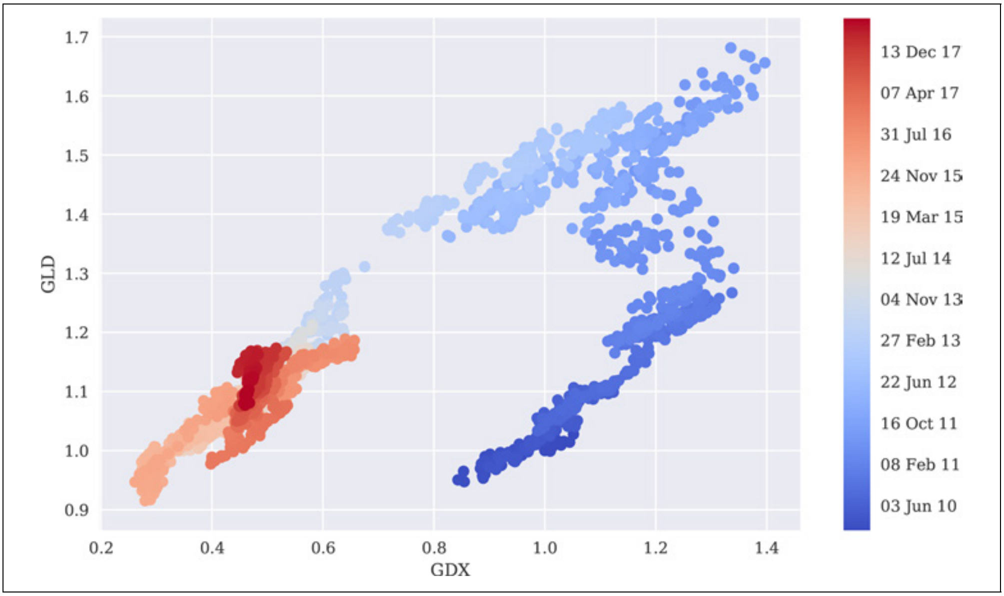


图 13-19　GLD 价格与 GDX 价格散点图

接下来在这两个时间序列基础上实施贝叶斯回归，参数化本质上和前一个虚拟数据示例的过程相同。图 13-20 展示了在 3 个参数先验概率分布的假设下，MCMC 采样过程的结果：

```
In [24]: with pm.Model() as model:
             alpha = pm.Normal('alpha', mu=0, sd=20)
             beta = pm.Normal('beta', mu=0, sd=20)
             sigma = pm.Uniform('sigma', lower=0, upper=50)
             y_est = alpha + beta * data['GDX'].values
```

```
        likelihood = pm.Normal('GLD', mu=y_est, sd=sigma,
                               observed=data['GLD'].values)

        start = pm.find_MAP()
        step = pm.NUTS()
        trace = pm.sample(250, tune=2000, start=start,
                          progressbar=True)
    logp = 1,493.7, ||grad|| = 188.29: 100%|█████| 27/27 [00:00<00:00,
     1609.34it/s]
    Only 250 samples in chain.
    Auto-assigning NUTS sampler...
    Initializing NUTS using jitter+adapt_diag...
    Multiprocess sampling (2 chains in 2 jobs)
    NUTS: [sigma, beta, alpha]
    Sampling 2 chains: 100%|█████| 4500/4500 [00:09<00:00,
     465.07draws/s]
    The estimated number of effective samples is smaller than 200 for some
     parameters.

In [25]: pm.summary(trace)
Out[25]:
             mean        sd  mc_error   hpd_2.5  hpd_97.5       n_eff      Rhat
    alpha 0.913335 0.005983 0.000356 0.901586 0.924714 184.264900 1.001855
     beta 0.385394 0.007746 0.000461 0.369154 0.398291 215.477738 1.001570
    sigma 0.119484 0.001964 0.000098 0.115305 0.123315 312.260213 1.005246

In [26]: fig = pm.traceplot(trace)
```

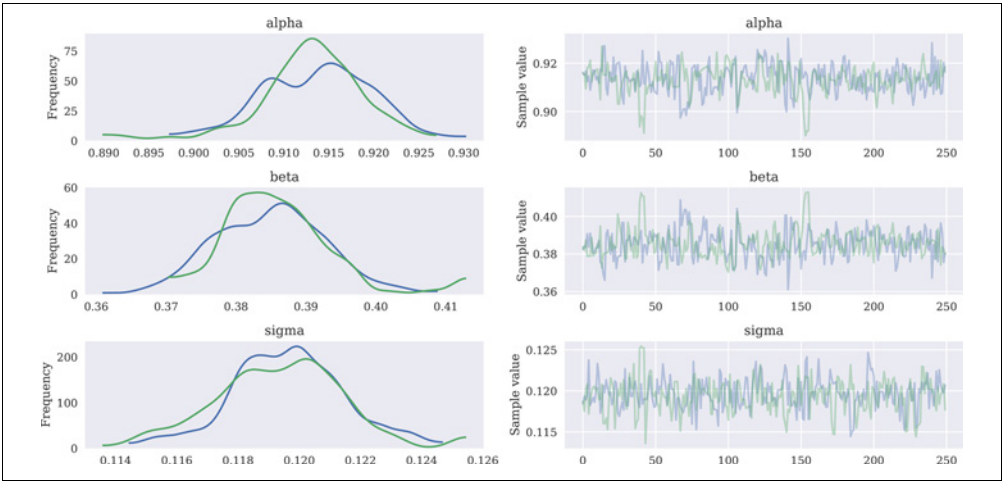


图 13-20　GDX 和 GLD 数据的后验分布及轨迹图

图 13-21 将得到的所有回归线添加到之前的散点图中。但是，所有回归线相互之间很靠近：

```
In [27]: plt.figure(figsize=(10, 6))
         plt.scatter(data['GDX'], data['GLD'], c=mpl_dates,
                     marker='o', cmap='coolwarm')
         plt.xlabel('GDX')
         plt.ylabel('GLD')
         for i in range(len(trace)):
             plt.plot(data['GDX'],
                      trace['alpha'][i] + trace['beta'][i] * data['GDX'])
         plt.colorbar(ticks=mpl.dates.DayLocator(interval=250),
                      format=mpl.dates.DateFormatter('%d %b %y'));
```

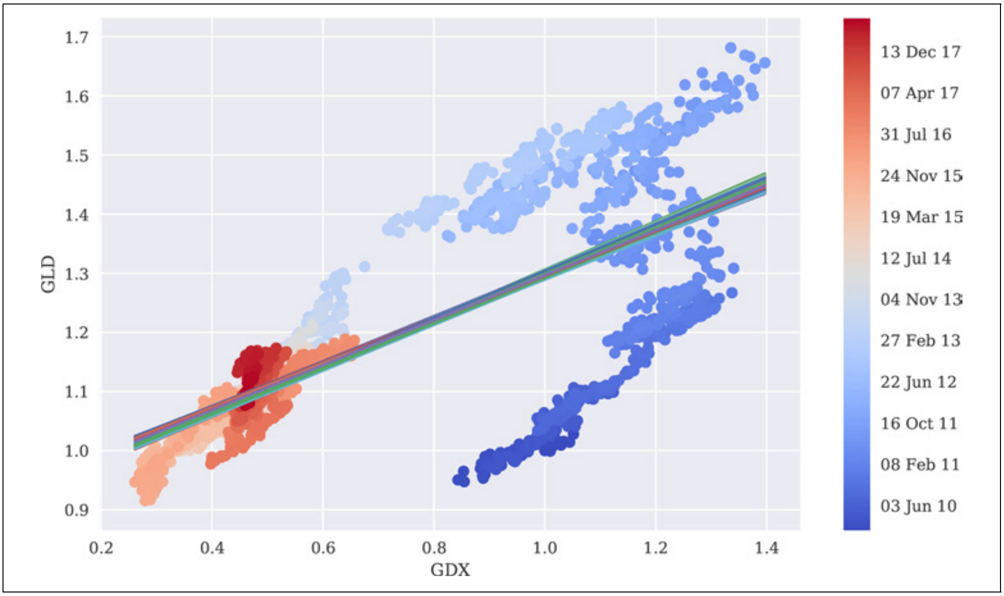


图 13-21　穿过 GDX 和 GLD 数据的多条贝叶斯回归线

图 13-21 揭示了所用回归方法的一个重大缺点：该方法没有考虑随时间推移发生的变化。也就是说，最近的数据和最老的数据受到同等对待。

13.3.4　随时更新估算值

正如前面指出的，金融学上的贝叶斯方法通常在历史分析时最有用——也就是说，随着时间推移揭示的新数据可以得出更好的回归和估算。

为了在当前的示例中加入上述概念，需要假定回归参数不只是随机和呈某种分布的，而且还会随着时间的推移随机“游走”。这和金融理论中从随机变量推广到随机过程（本质上是随机变量的有序序列）时采用了相同的方式。

为此，我们定义一个新的 PyMC3 模型，这次指定参数值随机游走。在指定随机游走参数的分布之后，我们可以继续指定随机游走的 `alpha` 和 `beta` 值。为了使整个过程更加高效，为 50 个数据点使用相同的系数：

```
In [28]: from pymc3.distributions.timeseries import GaussianRandomWalk

In [29]: subsample_alpha = 50
         subsample_beta = 50

In [30]: model_randomwalk = pm.Model()
         with model_randomwalk:
             sigma_alpha = pm.Exponential('sig_alpha', 1. / .02,
testval=.1) ❶
             sigma_beta = pm.Exponential('sig_beta', 1. / .02, testval=.1) ❶
             alpha = GaussianRandomWalk('alpha', sigma_alpha ** -2,
                                    shape=int(len(data) / subsample_alpha)) ❷
             beta = GaussianRandomWalk('beta', sigma_beta ** -2,
                                    shape=int(len(data) / subsample_beta)) ❷
             alpha_r = np.repeat(alpha, subsample_alpha) ❸
             beta_r = np.repeat(beta, subsample_beta) ❸
             regression = alpha_r + beta_r * data['GDX'].values[:2100] ❹
             sd = pm.Uniform('sd', 0, 20) ❺
             likelihood = pm.Normal('GLD', mu=regression, sd=sd,
                                    observed=data['GLD'].values[:2100]) ❻
```

❶ 为随机游走参数定义先验概率。

❷ 随机游走模型。

❸ 将参数向量代入时间间隔长度。

❹ 定义回归模型。

❺ 标准差的先验值。

❻ 用回归结果中的 `mu` 定义似然度。

由于使用随机游走代替单一随机变量，所以这些定义都比以前复杂一些。不过，MCMC 的推导步骤本质上仍然没有变化。要注意的是，由于我们必须为每个随机游走样本计算一个参数对，共计 1950/50=39 个（以前只需要 1 个），所以计算负担明显增大：

```
In [31]: %%time
         import scipy.optimize as sco
         with model_randomwalk:
             start = pm.find_MAP(vars=[alpha, beta],
                                    fmin=sco.fmin_l_bfgs_b)
             step = pm.NUTS(scaling=start)
```

```
    trace_rw = pm.sample(250, tune=1000, start=start,
                         progressbar=True)
logp = -6,657: 2%|         | 82/5000 [00:00<00:08, 550.29it/s]
Only 250 samples in chain.
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, beta, alpha, sig_beta, sig_alpha]
Sampling 2 chains: 100%|█████| 2500/2500 [02:48<00:00, 8.59draws/s]

CPU times: user 27.5 s, sys: 3.68 s, total: 31.2 s
Wall time: 5min 3s

In [32]: pm.summary(trace_rw).head() ❶
Out[32]:
             mean        sd  mc_error  hpd_2.5  hpd_97.5       n_eff \
    alpha__0 0.673846 0.040224 0.001376 0.592655 0.753034 1004.616544
    alpha__1 0.424819 0.041257 0.001618 0.348102 0.509757 804.760648
    alpha__2 0.456817 0.057200 0.002011 0.321125 0.553173 800.225916
    alpha__3 0.268148 0.044879 0.001725 0.182744 0.352197 724.967532
    alpha__4 0.651465 0.057472 0.002197 0.544076 0.761216 978.073246
             Rhat
    alpha__0 0.998637
    alpha__1 0.999540
    alpha__2 0.998075
    alpha__3 0.998995
    alpha__4 0.998060
```

❶ 每个时间间隔的汇总统计（仅显示前 5 个和 alpha）。

图 13-22 表现的是估算值的一个子集，说明回归参数 alpha 和 beta 随时间演变：

```
In [33]: sh = np.shape(trace_rw['alpha']) ❶
         sh ❶
Out[33]: (500, 42)

In [34]: part_dates = np.linspace(min(mpl_dates),
                         max(mpl_dates), sh[1]) ❷

In [35]: index = [dt.datetime.fromordinal(int(date)) for
                  date in part_dates] ❷

In [36]: alpha = {'alpha_%i' % i: v for i, v in
                  enumerate(trace_rw['alpha']) if i < 20} ❸

In [37]: beta = {'beta_%i' % i: v for i, v in
```

```
                    enumerate(trace_rw['beta']) if i < 20} ❸

In [38]: df_alpha = pd.DataFrame(alpha, index=index) ❸

In [39]: df_beta = pd.DataFrame(beta, index=index) ❸

In [40]: ax = df_alpha.plot(color='b', style='-.', legend=False,
                            lw=0.7, figsize=(10, 6))
         df_beta.plot(color='r', style='-.', legend=False,
                      lw=0.7, ax=ax)
         plt.ylabel('alpha/beta');
```

❶ 包含参数估算值的对象构成。

❷ 创建日期列表以匹配时间间隔数量。

❸ 将相关的参数时间序列收集在两个 DataFrame 对象中。

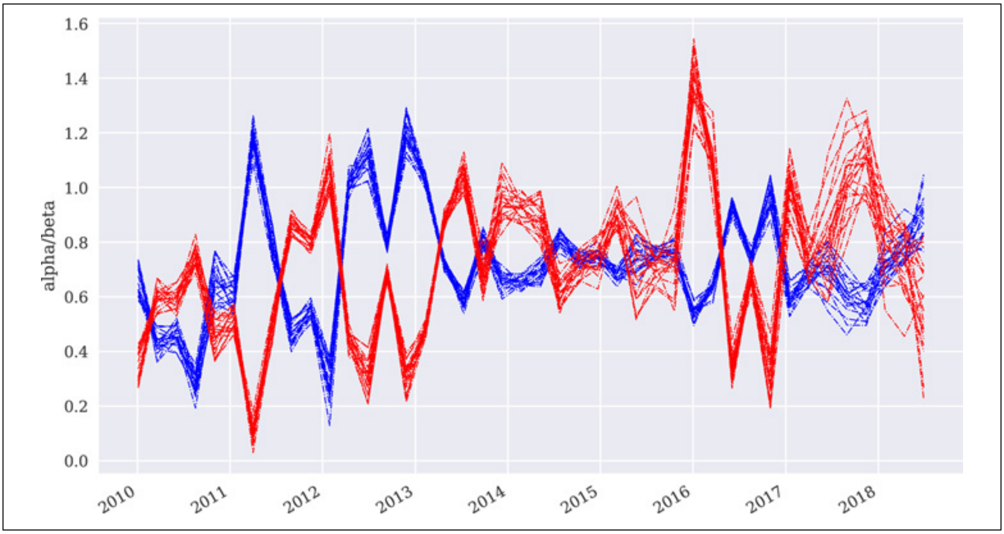


图 13-22　一段时间的选择参数估算值



绝对价格数据和相对收益率数据

本节的分析基于规范化价格数据。这仅仅是为了说明目的，因为对应的图形化结果更容易理解和解读（它们在视觉上“更引人注目”）。但是，现实世界中的金融应用应该依赖收益率数据，以确保时间序列数据的稳定性。

使用 alpha 和 beta 均值，我们可以说明一段时间内回归的更新。图 13-23 展示了回

归随时间推移而更新的情况。此外，它还显示了从 alpha 和 beta 均值得出的 39 条回归线。很明显，随时间推移而更新大大提高了回归拟合（对于当前/最新的数据）的精度——换言之，每个时间周期都需要自己的回归：

```
In [41]: plt.figure(figsize=(10, 6))
         plt.scatter(data['GDX'], data['GLD'], c=mpl_dates,
                     marker='o', cmap='coolwarm')
         plt.colorbar(ticks=mpl.dates.DayLocator(interval=250),
                      format=mpl.dates.DateFormatter('%d %b %y'))
         plt.xlabel('GDX')
         plt.ylabel('GLD')
         x = np.linspace(min(data['GDX']), max(data['GDX']))
         for i in range(sh[1]): ❶
             alpha_rw = np.mean(trace_rw['alpha'].T[i])
             beta_rw = np.mean(trace_rw['beta'].T[i])
             plt.plot(x, alpha_rw + beta_rw * x, '--', lw=0.7,
                     color=plt.cm.coolwarm(i / sh[1]))
```

❶ 为所有长度为 50 的时间间隔绘制回归线。

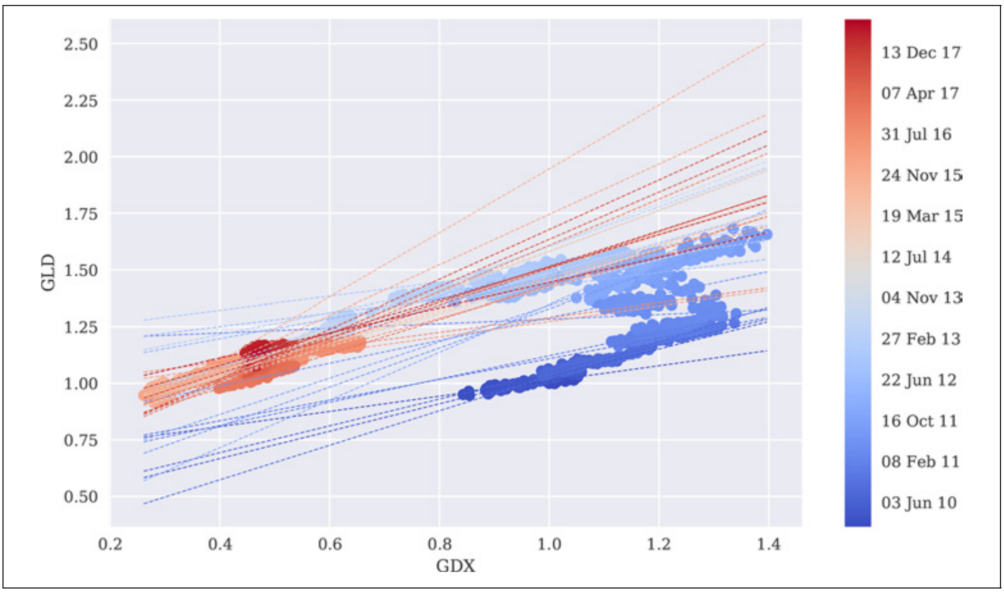


图 13-23　包含时间相关回归线（更新的估算）的散点图

关于贝叶斯回归的介绍到此告一段落，Python 提供了强大的 PyMC3 库，以实现贝叶斯统计以及概率编程中的不同方法，尤其是贝叶斯回归，它在计量金融学中已经成为相当流行而重要的工具。

13.4　机器学习

在金融和许多其他领域，机器学习（ML）是近年来非常重要的技术。正如：

> 计量经济学（目前来说）已经足以在金融学术界取得成功，但实践中的成功需要 ML。
>
> ——马科斯・洛佩斯・德普拉多（2018）

机器学习包括不同类型的算法，它们本质上可以从原始数据中自行学习某些关系、模式等。13.6 节中列出了许多图书，读者可以从中查阅关于机器学习方法及算法的数学和统计学特性，以及实现及实际使用的相关主题。例如，Alpaydin 的著作（2016）对这个领域做了很好的介绍，并对使用的典型算法类型做了非技术性的概述。

本节严格采用了实际的方法，只专注于有选择的实现方面——着眼于第 15 章中使用的技术。不过，我们介绍的算法和技术可以用于许多不同金融领域，不仅限于算法交易。本节介绍两类算法：无监督和有监督学习算法。

最流行的 Python 机器学习库之一是 scikit-learn。它不仅提供了许多 ML 算法的实现，还提供了大量有助于 ML 任务相关预处理和后处理活动的工具。本节主要依靠这个软件包。在深度神经网络（DNN）方面，本节还使用了 TensorFlow。

VanderPlas（2016）以 Python 和 scikit-learn 为基础，简练地介绍了不同的 ML 算法。Albon（2018）为 ML 典型任务提供了许多窍门，使用的也主要是 Python 和 scikit-learn。

13.4.1　无监督学习

无监督学习的思路是：机器学习算法在没有进一步指导的情况下，从原始数据中发现深层次的情况。*k*-均值（K-means）聚类就是这样的算法，它将原始数据集分为多个聚类，为这些子集分配标签（“聚类 0”“聚类 1”等）。另一个算法是高斯混合。[1]

1．数据

scikit-learn 可以为不同类型的 ML 问题创建样板数据。下面创建适合于参数 *k* 均值聚类的样板数据。

首先是一些标准的导入和配置：

```
In [1]: import numpy as np
        import pandas as pd
        import datetime as dt
        from pylab import mpl, plt
```

1　scikit-learn 中的更多无监督学习算法请参见 scikit-learn 官方文档。——原注

```
In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        np.random.seed(1000)
        np.set_printoptions(suppress=True, precision=4)
        %matplotlib inline
```

其次，创建样板数据集。图 13-24 可视化了这些样板数据：

```
In [3]: from sklearn.datasets.samples_generator import make_blobs

In [4]: X, y = make_blobs(n_samples=250, centers=4,
                          random_state=500, cluster_std=1.25) ❶

In [5]: plt.figure(figsize=(10, 6))
        plt.scatter(X[:, 0], X[:, 1], s=50);
```

❶ 创建用于聚类的样本数据集，包含 250 个样本和 4 个中心。

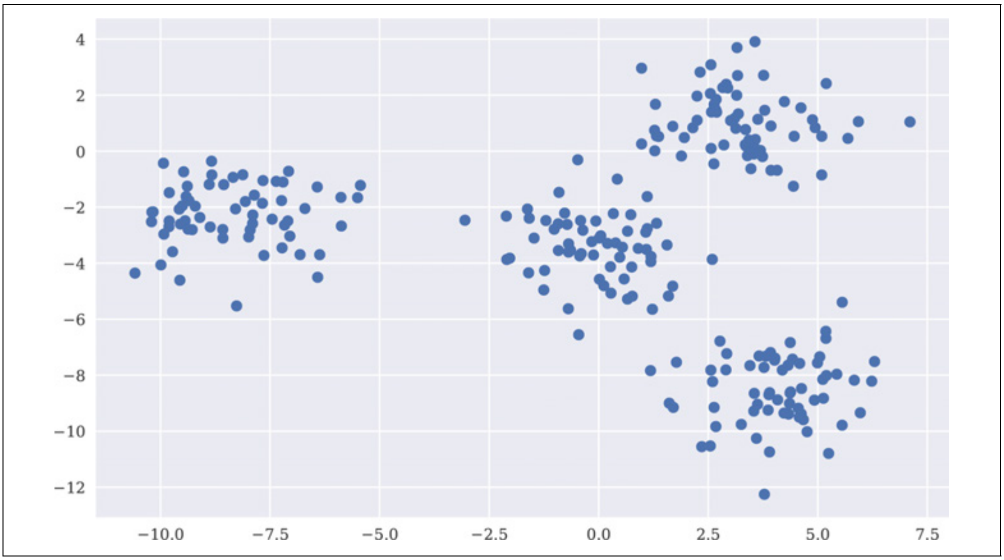


图 13-24　聚类算法应用的样板数据

2. *k* 均值聚类

scikit-learn 的特性之一是提供了应用不同类算法的标准化 API。以下代码展示了 *k* 均值聚类的基本步骤，此后的其他模型中也重复使用这些步骤。

- 导入模型类。
- 实例化模型对象。
- 将模型对象与某些数据拟合。

- 按照某些数据的拟合模型预测结果。

```
In [6]: from sklearn.cluster import KMeans ❶

In [7]: model = KMeans(n_clusters=4, random_state=0) ❷

In [8]: model.fit(X) ❸
Out[8]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
               n_clusters=4, n_init=10, n_jobs=None, precompute_distances='auto',
               random_state=0, tol=0.0001, verbose=0)

In [9]: y_kmeans = model.predict(X) ❹

In [10]: y_kmeans[:12] ❺
Out[10]: array([1, 1, 0, 3, 0, 1, 3, 3, 3, 0, 2, 2], dtype=int32)

In [11]: plt.figure(figsize=(10, 6))
         plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='coolwarm');
```

❶ 从 scikit-learn 中导入模型类。

❷ 按照某些参数实例化模型对象，在实例化中使用关于样本数据的知识。

❸ 将模型对象与原始数据拟合。

❹ 根据原始数据预测聚类（编号）。

❺ 显示预测的聚类编号。

结果如图 13-25 所示。



图 13-25　样本数据与识别的聚类

3. 高斯混合

需要替代的聚类方法时，可以考虑高斯混合。应用方法与 k 均值相同。在参数合适时，结果也一样：

```
In [12]: from sklearn.mixture import GaussianMixture

In [13]: model = GaussianMixture(n_components=4, random_state=0)

In [14]: model.fit(X)
Out[14]: GaussianMixture(covariance_type='full', init_params='kmeans',
          max_iter=100,
         means_init=None, n_components=4, n_init=1, precisions_init=None,
                 random_state=0, reg_covar=1e-06, tol=0.001, verbose=0,
                 verbose_interval=10, warm_start=False, weights_init=None)

In [15]: y_gm = model.predict(X)

In [16]: y_gm[:12]
Out[16]: array([1, 1, 0, 3, 0, 1, 3, 3, 3, 0, 2, 2])

In [17]: (y_gm == y_kmeans).all() ❶
Out[17]: True
```

❶ k 均值聚类与高斯混合的结果相同。

13.4.2　有监督学习

有监督学习是具有某种指导（形式是已知的结果或者观测数据）的机器学习方法。这意味着，原始数据中已经包含了 ML 算法打算学习的东西。下面的讨论重点是分类问题而非估算问题。估算问题通常是估算实际数量，而分类问题的特征是为某一特征组合分配相对小的类集（整数值）中的某个类（整数值）。

13.4.1 节中的例子说明，利用无监督学习，算法可以为识别的聚类打上分类标签。例中有 4 个聚类，标签分别为 0、1、2 和 3。有监督学习中的分类标签已经给出，因此算法可以学习特征与类别（分类）之间的关系。换言之，在拟合步骤中，算法知道给定特征值组合的正确分类。

本节将阐述这些分类算法：高斯朴素贝叶斯、逻辑回归、决策树、深度神经网络和支持向量机的应用。[1]

1　scikit-learn 用于有监督学习的分类算法概述请参见 scikit-learn 文档。注意，许多此类算法不仅可用于分类，也可用于估算。——原注

1. 数据

scikit-learn 同样可以为应用分类算法创建合适的样本数据。为了可视化结果，样本数据只包含两个实值信息特征和一个二元标签（二元标签的特点是仅以 0、1 区分两个不同类）。下面的代码用于创建样本数据、显示数据摘要并对其进行可视化（结果见图 13-26）：

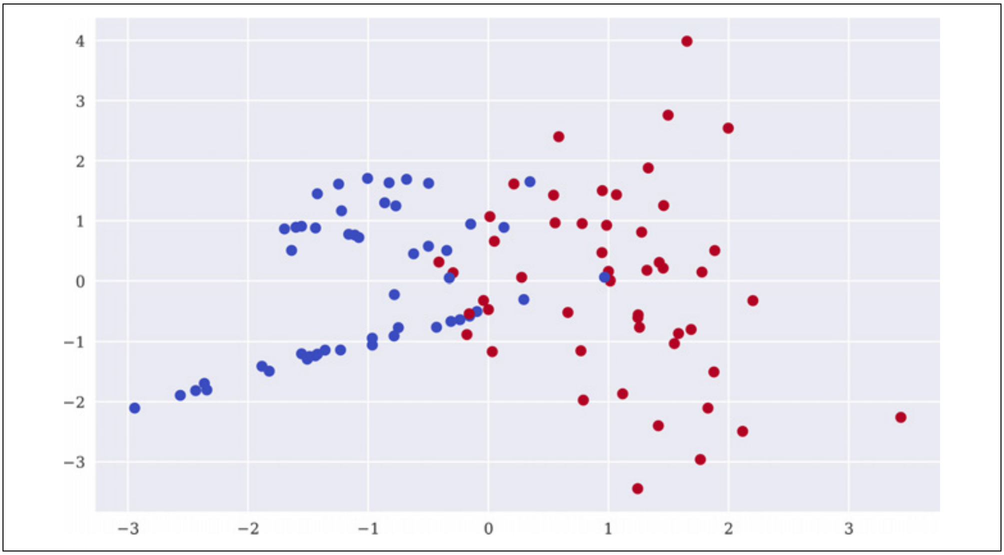


图 13-26　分类算法应用的样本数据

```
In [18]: from sklearn.datasets import make_classification

In [19]: n_samples = 100

In [20]: X, y = make_classification(n_samples=n_samples, n_features=2,
                                    n_informative=2, n_redundant=0,
                                    n_repeated=0, random_state=250)

In [21]: X[:5] ❶
Out[21]: array([[ 1.6876, -0.7976],
                [-0.4312, -0.7606],
                [-1.4393, -1.2363],
                [ 1.118 , -1.8682],
                [ 0.0502, 0.659 ]])
In [22]: X.shape ❶
Out[22]: (100, 2)

In [23]: y[:5] ❷
Out[23]: array([1, 0, 0, 1, 1])
```

```
In [24]: y.shape ❷
Out[24]: (100,)

plt.figure(figsize=(10, 6))
plt.hist(X);
In [25]: plt.figure(figsize=(10, 6))
         plt.scatter(x=X[:, 0], y=X[:, 1], c=y, cmap='coolwarm');
```

❶ 两个实值信息特征。

❷ 二元标签。

2. 高斯朴素贝叶斯

高斯朴素贝叶斯（GNB）一般被看作许多分类问题中很好的基准算法。其应用与 13.4.1 节中的 *R* 均值聚类部分所介绍的相同：

```
In [26]: from sklearn.naive_bayes import GaussianNB
         from sklearn.metrics import accuracy_score
In [27]: model = GaussianNB()

In [28]: model.fit(X, y)
Out[28]: GaussianNB(priors=None, var_smoothing=1e-09)

In [29]: model.predict_proba(X).round(4)[:5] ❶
Out[29]: array([[0.0041, 0.9959],
                [0.8534, 0.1466],
                [0.9947, 0.0053],
                [0.0182, 0.9818],
                [0.5156, 0.4844]])

In [30]: pred = model.predict(X) ❷

In [31]: pred ❷
Out[31]: array([1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1,
          0,
         0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0,
         0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0,
         0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,
         0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0])

In [32]: pred == y ❸
Out[32]: array([ True, True, True, True, False, True, True, True, True,
                 True, False, True, True, True, True, True, True, True,
                 True, True, True, True, False, False, False, True, True,
                 True, True, True, True, True, True, False, True, True,
```

```
              True, True, True, True, True, True, True, True, True,
              True, True, True, True, True, True, False, True, False,
              True, True, True, True, True, True, True, True, True,
              True, True, False, True, True, True, True, True, True,
              True, True, True, True, True, True, False, True, False,
              True, True, True, True, True, True, True, True, True,
              True, True, False, True, False, True, True, True, True,
              True])
```

```
In [33]: accuracy_score(y, pred) ❹
Out[33]: 0.87
```

❶ 显示算法在拟合之后为每个类指定的概率。

❷ 根据概率来预测数据集的两个分类。

❸ 比较预测分类与实际分类。

❹ 按照预测值计算准确率。

图 13-27 可视化了 GNB 中正确和错误的预测，代码如下。

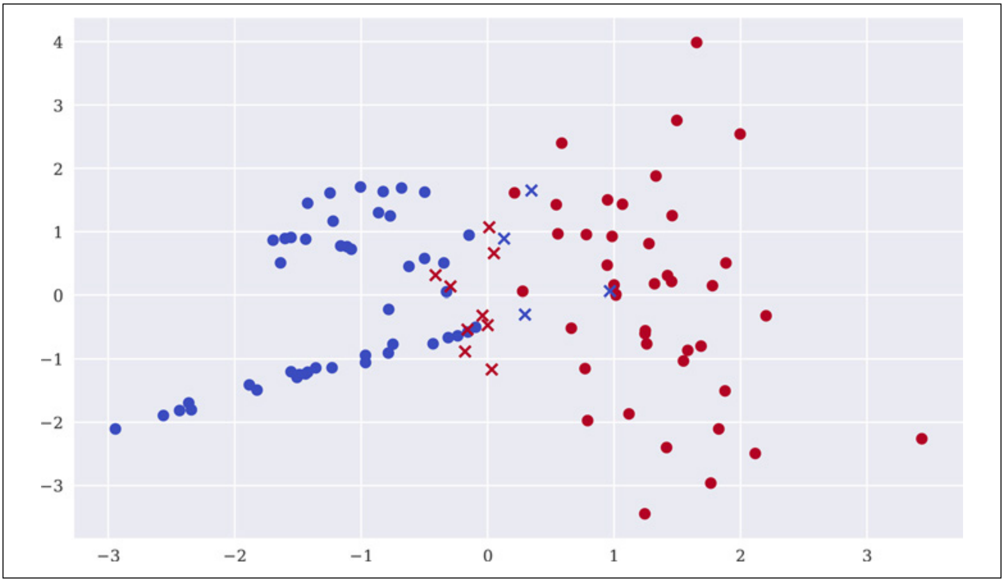


图 13-27　GNB 的正确（点）和错误（叉）预测

```
In [34]: Xc = X[y == pred] ❶
         Xf = X[y != pred] ❷

In [35]: plt.figure(figsize=(10, 6))
```

```
plt.scatter(x=Xc[:, 0], y=Xc[:, 1], c=y[y == pred],
            marker='o', cmap='coolwarm') ❶
plt.scatter(x=Xf[:, 0], y=Xf[:, 1], c=y[y != pred],
            marker='x', cmap='coolwarm') ❷
```

❶ 选择正确的预测并绘制图表。

❷ 选择错误的预测并绘制图表。

3. 逻辑回归

逻辑回归（LR）是一种快速、可缩放的分类算法。在以下示例中，其精确度略好于 GNB：

```
In [36]: from sklearn.linear_model import LogisticRegression

In [37]: model = LogisticRegression(C=1, solver='lbfgs')

In [38]: model.fit(X, y)
Out[38]: LogisticRegression(C=1, class_weight=None, dual=False,
          fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='warn',
                   n_jobs=None, penalty='l2', random_state=None,
                   solver='lbfgs',tol=0.0001, verbose=0, warm_start=False)

In [39]: model.predict_proba(X).round(4)[:5]
Out[39]: array([[0.011 , 0.989 ],
                [0.7266, 0.2734],
                [0.971 , 0.029 ],
                [0.04  , 0.96  ],
                [0.4843, 0.5157]])

In [40]: pred = model.predict(X)

In [41]: accuracy_score(y, pred)
Out[41]: 0.9

In [42]: Xc = X[y == pred]
         Xf = X[y != pred]

In [43]: plt.figure(figsize=(10, 6))
         plt.scatter(x=Xc[:, 0], y=Xc[:, 1], c=y[y == pred],
                     marker='o', cmap='coolwarm')
         plt.scatter(x=Xf[:, 0], y=Xf[:, 1], c=y[y != pred],
                     marker='x', cmap='coolwarm');
```

4. 决策树

决策树（DT）是另一类很容易缩放的分类算法。在最大深度为 1 的情况下，这种算法的表现已经略好于 GNB 和 LR（见图 13-28）：

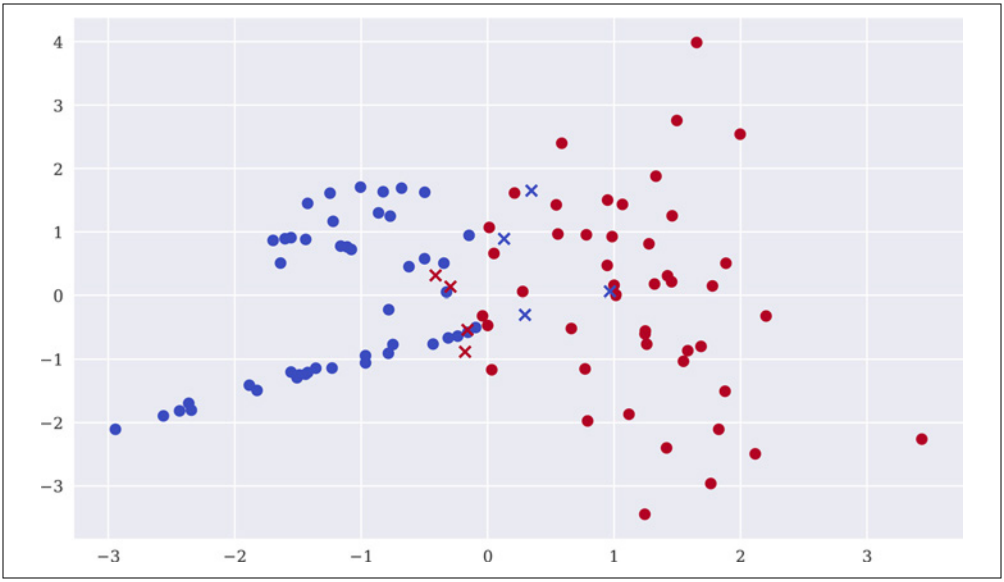


图 13-28　DT 中的正确（点）与错误（叉）预测（最大深度为 1）

```
In [44]: from sklearn.tree import DecisionTreeClassifier

In [45]: model = DecisionTreeClassifier(max_depth=1)

In [46]: model.fit(X, y)
Out[46]: DecisionTreeClassifier(class_weight=None, criterion='gini',
          max_depth=1,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
         min_weight_fraction_leaf=0.0, presort=False, random_state=None,
                      splitter='best')

In [47]: model.predict_proba(X).round(4)[:5]
Out[47]: array([[0.08, 0.92],
                [0.92, 0.08],
                [0.92, 0.08],
                [0.08, 0.92],
                [0.08, 0.92]])
```

```
In [48]: pred = model.predict(X)

In [49]: accuracy_score(y, pred)
Out[49]: 0.92

In [50]: Xc = X[y == pred]
         Xf = X[y != pred]

In [51]: plt.figure(figsize=(10, 6))
         plt.scatter(x=Xc[:, 0], y=Xc[:, 1], c=y[y == pred],
                     marker='o', cmap='coolwarm')
         plt.scatter(x=Xf[:, 0], y=Xf[:, 1], c=y[y != pred],
                     marker='x', cmap='coolwarm');
```

增大决策树的最大深度参数，可以达到完美的结果：

```
In [52]: print('{:>8s} | {:8s}'.format('depth', 'accuracy'))
         print(20 * '-')
         for depth in range(1, 7):
             model = DecisionTreeClassifier(max_depth=depth)
             model.fit(X, y)
             acc = accuracy_score(y, model.predict(X))
             print('{:8d} | {:8.2f}'.format(depth, acc))
            depth | accuracy
         --------------------
                1 |     0.92
                2 |     0.92
                3 |     0.94
                4 |     0.97
                5 |     0.99
                6 |     1.00
```

5. 深度神经网络

深度神经网络（DNN）被认为是最强大但计算要求也很高的估算与分类算法之一。它的流行部分归功于 Google 开放的 TensorFlow 库源码及其成功事迹。DNN 可以学习复杂的非线性关系并为之建模。虽然它起源的时间可以追溯到 20 世纪 70 年代，但直到最近几年，随着硬件（CPU、GPU、TPU）、数值算法及相关软件实现的进步，它才得以大规模应用。

其他 ML 算法（如 LR 类线性模型）可以根据标准优化问题高效拟合，而 DNN 依赖于深度学习，这通常需要大量重复步骤以调整某些参数（权重），从而比较结果与数据。从这个意义上说，深度学习可以和金融数学中的蒙特卡洛模拟相提并论，例如，欧式看涨期权的价格估算可能在标的的 10 万条模拟路径基础上进行。另一方面，Black-

Scholes-Merton 期权定价公式采用的是封闭形式，可以进行分析性的评估。

蒙特卡洛模拟是金融数学中最为灵活和强大的数字技术之一，但代价是高计算负担和大内存占用。深度学习也是如此，它通常比许多其他 ML 算法更灵活，但它需要的计算能力也更大。

用 scikit-learn 实现 DNN。虽然性质上有很大的不同，但 scikit-learn 为其 MLPClassifier 算法类提供了相同的 API。该类是一个 DNN 模型，可用于此前使用的其他 ML 算法。仅用两个所谓的隐含层，它就能在测试数据上得到完美的结果（隐含层是深度学习有别于简单学习的地方——例如，在线性回归背景下“学习”权重，而不是使用 OLS 回归直接得出它们）：

```
In [53]: from sklearn.neural_network import MLPClassifier

In [54]: model = MLPClassifier(solver='lbfgs', alpha=1e-5,
                               hidden_layer_sizes=2 * [75], random_state=10)

In [55]: %time model.fit(X, y)
         CPU times: user 537 ms, sys: 14.2 ms, total: 551 ms
         Wall time: 340 ms

Out[55]: MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
          beta_1=0.9,
                beta_2=0.999, early_stopping=False, epsilon=1e-08,
                hidden_layer_sizes=[75, 75], learning_rate='constant',
                learning_rate_init=0.001, max_iter=200, momentum=0.9,
                n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
                random_state=10, shuffle=True, solver='lbfgs', tol=0.0001,
                validation_fraction=0.1, verbose=False, warm_start=False)

In [56]: pred = model.predict(X)
         pred
Out[56]: array([1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1,
          0,
         1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0,
         0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1,
         0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1,
               0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0])

In [57]: accuracy_score(y, pred)
Out[57]: 1.0
```

用 TensorFlow 实现 DNN。TensorFlow 的 API 与 scikit-learn 标准不同。但是，`DNNClassifier` 类的应用同样简单：

```
In [58]: import tensorflow as tf
         tf.logging.set_verbosity(tf.logging.ERROR) ❶

In [59]: fc = [tf.contrib.layers.real_valued_column('features')] ❷

In [60]: model = tf.contrib.learn.DNNClassifier(hidden_units=5 * [250],
                                                n_classes=2,
                                                feature_columns=fc) ❸

In [61]: def input_fn(): ❹
             fc = {'features': tf.constant(X)}
             la = tf.constant(y)
             return fc, la

In [62]: %time model.fit(input_fn=input_fn, steps=100) ❺
         CPU times: user 7.1 s, sys: 1.35 s, total: 8.45 s
         Wall time: 4.71 s

Out[62]: DNNClassifier(params={'head':
         <tensorflow.contrib.learn.python.learn ... head._BinaryLogisticHead
         object at 0x1a3ee692b0>, 'hidden_units': [250, 250, 250, 250, 250],
         'feature_columns': (_RealValuedColumn(column_name='features',
         dimension=1, default_value=None, dtype=tf.float32, normalizer=
None),),
         'optimizer': None, 'activation_fn': <function relu at 0x1a3aa75b70>,
         'dropout': None, 'gradient_clip_norm': None,
         'embedding_lr_multipliers': None, 'input_layer_min_slice_
size': None})

In [63]: model.evaluate(input_fn=input_fn, steps=1) ❺
Out[63]: {'loss': 0.18724777,
          'accuracy': 0.91,
          'labels/prediction_mean': 0.5003989,
          'labels/actual_label_mean': 0.5,
          'accuracy/baseline_label_mean': 0.5,
          'auc': 0.9782,
          'auc_precision_recall': 0.97817385,
          'accuracy/threshold_0.500000_mean': 0.91,
          'precision/positive_threshold_0.500000_mean': 0.9019608,
          'recall/positive_threshold_0.500000_mean': 0.92,
          'global_step': 100}

In [64]: pred = np.array(list(model.predict(input_fn=input_fn))) ❻
         pred[:10] ❻
Out[64]: array([1, 0, 0, 1, 1, 0, 1, 1, 1, 1])
```

```
In [65]: %time model.fit(input_fn=input_fn, steps=750) ❼
         CPU times: user 29.8 s, sys: 7.51 s, total: 37.3 s
         Wall time: 13.6 s

Out[65]: DNNClassifier(params={'head':
         <tensorflow.contrib.learn.python.learn ... head._BinaryLogisticHead
         object at 0x1a3ee692b0>, 'hidden_units': [250, 250, 250, 250, 250],
         'feature_columns': (_RealValuedColumn(column_name='features',
         dimension=1, default_value=None, dtype=tf.float32, normalizer=
None),),
         'optimizer': None, 'activation_fn': <function relu at 0x1a3aa75b70>,
         'dropout': None, 'gradient_clip_norm': None,
         'embedding_lr_multipliers': None, 'input_layer_min_slice_
size': None})

In [66]: model.evaluate(input_fn=input_fn, steps=1) ❽
Out[66]: {'loss': 0.09271307,
          'accuracy': 0.94,
          'labels/prediction_mean': 0.5274486,
          'labels/actual_label_mean': 0.5,
          'accuracy/baseline_label_mean': 0.5,
          'auc': 0.99759996,
          'auc_precision_recall': 0.9977609,
          'accuracy/threshold_0.500000_mean': 0.94,
          'precision/positive_threshold_0.500000_mean': 0.9074074,
          'recall/positive_threshold_0.500000_mean': 0.98,
          'global_step': 850}
```

❶ 设置 TensorFlow 日志的详细程度。

❷ 抽象定义实值特征。

❸ 实例化模型对象。

❹ 特征和标签数据由函数提交。

❺ 通过学习拟合模型并评估。

❻ 根据特征值预测标签值。

❼ 以更多学习步骤为基础重新训练模型；前面的结果作为起始点。

❽ 重新训练后精度提高。

这里介绍的只是 TensorFlow 的一点皮毛，它被用于许多高要求的用例中，例如 Alphabet 公司制造自动驾驶汽车。就速度而言，TensorFlow 模型的训练通常显著得益于专用硬件（如 GPU 和 TPU），而不是 CPU。

6. 特征空间变换

由于多种原因，变换实值特征可能有好处，甚至是必要的。下面的代码展示了一些典型的变化，并在图 13-29 中可视化结果以进行比较：

```
In [67]: from sklearn import preprocessing

In [68]: X[:5]
Out[68]: array([[ 1.6876, -0.7976],
                [-0.4312, -0.7606],
                [-1.4393, -1.2363],
                [ 1.118 , -1.8682],
                [ 0.0502,  0.659 ]])

In [69]: Xs = preprocessing.StandardScaler().fit_transform(X) ❶
         Xs[:5]
Out[69]: array([[ 1.2881, -0.5489],
                [-0.3384, -0.5216],
                [-1.1122, -0.873 ],
                [ 0.8509, -1.3399],
                [ 0.0312,  0.5273]])

In [70]: Xm = preprocessing.MinMaxScaler().fit_transform(X) ❷
         Xm[:5]
Out[70]: array([[0.7262, 0.3563],
                [0.3939, 0.3613],
                [0.2358, 0.2973],
                [0.6369, 0.2122],
                [0.4694, 0.5523]])

In [71]: Xn1 = preprocessing.Normalizer(norm='l1').transform(X) ❸
         Xn1[:5]
Out[71]: array([[ 0.6791, -0.3209],
                [-0.3618, -0.6382],
                [-0.5379, -0.4621],
                [ 0.3744, -0.6256],
                [ 0.0708,  0.9292]])

n [72]: Xn2 = preprocessing.Normalizer(norm='l2').transform(X) ❸
        Xn2[:5]
Out[72]: array([[ 0.9041, -0.4273],
                [-0.4932, -0.8699],
                [-0.7586, -0.6516],
                [ 0.5135, -0.8581],
                [ 0.076 ,  0.9971]])
```

```
In [73]: plt.figure(figsize=(10, 6))
         markers = ['o', '.', 'x', '^', 'v']
         data_sets = [X, Xs, Xm, Xn1, Xn2]
         labels = ['raw', 'standard', 'minmax', 'norm(1)', 'norm(2)']
         for x, m, l in zip(data_sets, markers, labels):
             plt.scatter(x=x[:, 0], y=x[:, 1], c=y,
                         marker=m, cmap='coolwarm', label=l)
         plt.legend();
```

❶ 将特征数据变换为均值为 0、单位方差的标准正态分布数据。

❷ 将特征数据变换为各个特征的给定范围，具体范围由特征的最小值和最大值定义。

❸ 将特征数据比例单独调整为单位规范（L1 或者 L2）。

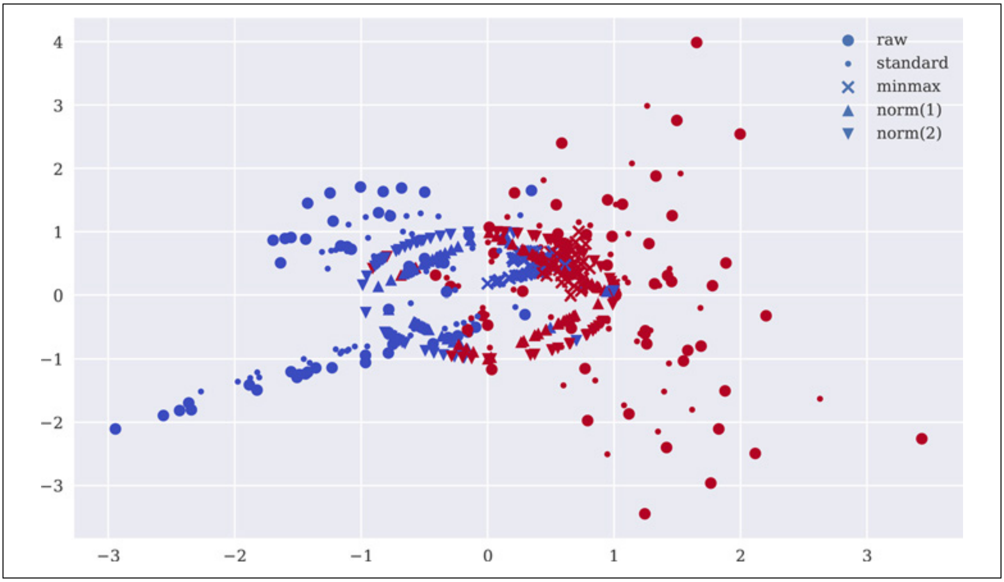


图 13-29　原始数据与变换后数据的对比

对于模式识别任务，分类特征的变换往往很有帮助，甚至是实现可接受结果所必需的。为此，特征的实值需要映射到有限、固定数量的整数值（类别、分类）：

```
In [74]: X[:5]
Out[74]: array([[ 1.6876, -0.7976],
                [-0.4312, -0.7606],
                [-1.4393, -1.2363],
                [ 1.118 , -1.8682],
                [ 0.0502, 0.659 ]])
```

```
In [75]: Xb = preprocessing.Binarizer().fit_transform(X) ❶
         Xb[:5]
Out[75]: array([[1., 0.],
                [0., 0.],
                [0., 0.],
                [1., 0.],
                [1., 1.]])

In [76]: 2 ** 2 ❷
Out[76]: 4

In [77]: Xd = np.digitize(X, bins=[-1, 0, 1]) ❸
         Xd[:5]
Out[77]: array([[3, 1],
                [1, 1],
                [0, 0],
                [3, 0],
                [2, 2]])

In [78]: 4 ** 2 ❹
Out[78]: 16
```

❶ 将特征变换为二元特征。

❷ 两个二元特征的可能特征值组合数量。

❸ 根据用于数据分组的值列表，将特征变换为分类特征。

❹ 可能特征值组合数量，两个特征各有 3 个值用于分组。

7. 训练-测试分离：支持向量机

现在，每个有经验的 ML 研究人员和从业者看到本书，都可能会关心本节的实现：它们都依靠同样的数据进行训练、学习和预测。如果将不同数据（子）集用于训练、学习以及测试，当然可以更好地评判 ML 算法的质量，这也更接近于现实应用场景。

scikit-learn 也提供了一个函数来高效地实现这种方法。特别是，`train_test_split()` 函数可以通过随机但可重复的方式将数据集拆分为训练数据和测试数据。

下面的代码使用了另一种分类算法——支持向量机（SVM）。首先根据训练数据拟合 SVM 模型：

```
In [79]: from sklearn.svm import SVC
         from sklearn.model_selection import train_test_split

In [80]: train_x, test_x, train_y, test_y = train_test_split(X, y,
test_size=0.33,
```

```
                                                    random_state=0)

    In [81]: model = SVC(C=1, kernel='linear')

    In [82]: model.fit(train_x, train_y) ❶
    Out[82]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
                 decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
               kernel='linear', max_iter=-1, probability=False, random_state=None,
               shrinking=True, tol=0.001, verbose=False)

    In [83]: pred_train = model.predict(train_x) ❷

    In [84]: accuracy_score(train_y, pred_train) ❸
    Out[84]: 0.9402985074626866
```

❶ 根据训练数据拟合模型。

❷ 预测训练数据标签值。

❸ 训练数据预测精度（“样本中”）。

接下来，根据测试数据测试拟合模型。图 13-30 展示了测试数据的正确和错误预测。在测试数据上的准确率正如人们的预期——低于训练数据：

```
    In [85]: pred_test = model.predict(test_x) ❶

    In [86]: test_y == pred_test ❷
    Out[86]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                     True, False, False, False,  True,  True,  True, False, False,
                    False,  True,  True,  True,  True,  True,  True,  True,  True,
                     True,  True,  True,  True, False,  True])

    In [87]: accuracy_score(test_y, pred_test) ❷
    Out[87]: 0.7878787878787878

    In [88]: test_c = test_x[test_y == pred_test]
             test_f = test_x[test_y != pred_test]

    In [89]: plt.figure(figsize=(10, 6))
             plt.scatter(x=test_c[:, 0], y=test_c[:, 1], c=test_y[test_y
 == pred_test],
                         marker='o', cmap='coolwarm')
             plt.scatter(x=test_f[:, 0], y=test_f[:, 1], c=test_y[test_y != p
red_test],
                         marker='x', cmap='coolwarm');
```

❶ 根据测试数据预测测试数据标签值。

❷ 求得拟合模型在测试数据上的准确率（“样本外”）。

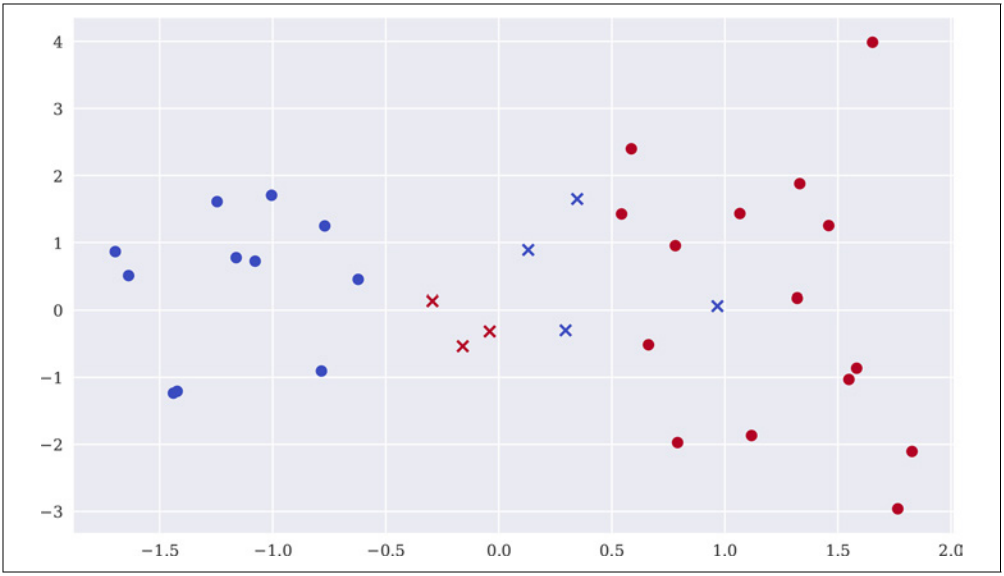


图 13-30 SVM 在测试数据上的正确（点）和错误（叉）预测

SVM 分类算法提供了所使用核心的多种选项。从以下分析可以看出，根据手头的问题，不同核心可能导致大不相同的结果（即不同的准确率）。这段代码首先将实值特征变换为分类特征：

```
In [90]: bins = np.linspace(-4.5, 4.5, 50)

In [91]: Xd = np.digitize(X, bins=bins)

In [92]: Xd[:5]
Out[92]: array([[34, 21],
                [23, 21],
                [17, 18],
                [31, 15],
                [25, 29]])

In [93]: train_x, test_x, train_y, test_y = train_test_split(Xd, y,
test_size=0.33,
                                                           random_state=0)

In [94]: print('{:>8s} | {:8s}'.format('kernel', 'accuracy'))
         print(20 * '-')
         for kernel in ['linear', 'poly', 'rbf', 'sigmoid']:
             model = SVC(C=1, kernel=kernel, gamma='auto')
```

```
    model.fit(train_x, train_y)
    acc = accuracy_score(test_y, model.predict(test_x))
    print('{:>8s} | {:8.3f}'.format(kernel, acc))
  kernel | accuracy
--------------------
  linear |    0.848
    poly |    0.758
     rbf |    0.788
 sigmoid |    0.455
```

13.5　结语

统计学不仅本身是个重要学科，还为许多其他学科（如金融学和社会科学）提供了不可或缺的工具。在一个章节中完整地讲解这么大的主题是不可能的。因此，本章专注于 4 个主题，以实例为基础阐述了 Python 和多种统计库的使用。

正态性

关于金融市场收益的正态性假设对许多金融理论和应用非常重要；因此，检验某些时间序列数据是否遵循这一假设也很重要。正如 13.1 节通过图形和统计学手段表明的那样，现实世界的收益率数据通常不遵循正态分布。

投资组合优化

MPT 的焦点是收益率的均值和方差/波动率，它被视为金融统计学概念和智能方面的重大成功。分散投资这一重要概念在这一背景中得到了很好的诠释。

贝叶斯回归

贝叶斯统计（尤其是贝叶斯回归）已经成为金融学中的流行工具，因为这种方法克服了第 11 章介绍的其他方法的缺点。尽管它的数学和形式体系更加复杂，但是基本思路（如随时间推移更新概率/分布）很容易直观地掌握。

机器学习

当今，机器学习已经和传统统计学方法与技术一样，确立了在金融领域中的地位。本章介绍了无监督学习（如 k 均值聚类）和有监督学习（如 DNN 分类器）ML 算法，并阐述了一些精选的相关主题，如特征变换及训练-测试分离。

13.6　延伸阅读

关于本章介绍的主题和库的更多信息，可参考以下在线资源：

① SciPy 统计学函数文档；

② statsmodels 库文档；

③ 本章使用的优化函数细节详见 SciPy 官方文档的 Optimization and Root Finding 部分；

④ PyMC3 文档；

⑤ scikit-learn 文档。

更多背景信息相关的参考图书如下。

- Albon, Chris (2018). *Machine Learning with Python Cookbook*. Sebastopol, CA: O'Reilly.
- Alpaydin, Ethem (2016). *Machine Learning*. Cambridge, MA: MIT Press.
- Copeland, Thomas, Fred Weston, and Kuldeep Shastri (2005). *Financial Theory and Corporate Policy*. Boston, MA: Pearson.
- Downey, Allen (2013). *Think Bayes*. Sebastopol, CA: O'Reilly.
- Geweke, John (2005). *Contemporary Bayesian Econometrics and Statistics*. Hoboken, NJ: John Wiley & Sons.
- Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer.
- James, Gareth, et al. (2013). *An Introduction to Statistical Learning—With Applications in R*. New York: Springer.
- Lopez de Prado, Marcos (2018). *Advances in Financial Machine Learning*. Hoboken, NJ: John Wiley & Sons.
- Rachev, Svetlozar, et al. (2008). *Bayesian Methods in Finance*. Hoboken, NJ: John Wiley & Sons.
- VanderPlas, Jake (2016). *Python Data Science Handbook*. Sebastopol, CA: O'Reilly.

介绍现代投资组合理论的论文有：

Markowitz, Harry (1952). “Portfolio Selection.” Journal of Finance, Vol. 7, pp. 77-91.

第 4 部分

算法交易

第 4 部分介绍 Python 在算法交易上的应用。越来越多的交易平台和经纪商允许客户使用各种工具（如 REST API），以编程的方式读取历史数据或流数据，或者下单买入及卖出。大型金融机构长期以来占据的这块“领地”，现在居然向使用算法交易的散户敞开了大门。除了其他因素之外，主要推动因素是许多交易平台（如福汇集团）为其 REST API 提供了易于使用的 Python 包装器库。

第 4 部分由 3 章组成。

- 第 14 章介绍福汇（FXCM）交易平台、它的 REST API 和 fxcmpy 包装器库。
- 第 15 章的重点是使用统计学及机器学习方法得出算法交易策略；本章还介绍如何使用向量化来进行事后检验。
- 第 16 章介绍自动化算法交易策略的部署；包括资产管理、绩效与风险事后检验、在线算法及部署。

第 14 章
FXCM 交易平台

金融机构喜欢把自己所做的事情称作交易。说老实话，那不是交易，那是赌博。

——格雷顿 • 卡特

本章介绍福汇集团（FXCM Group，LLC）的交易平台、它的 REST 风格流应用编程接口（API）以及 Python 包装器库 fxcmpy。FXCM 向散户和机构投资者提供了许多金融产品，它们既可以通过传统交易应用程序交易，也可以通过 API 编程交易。这些产品的焦点是货币组合以及主要股票指数及商品等的差价和约。

风险提示

外汇/CFD 保证金交易的风险很高，有可能遭受超出保证金额度的损失，因此不一定适合于所有投资者。杠杆可能对你产生不利影响。这些产品面向散户和专业客户。由于地方法规与监管的某些限制，居住在德国的散户可能损失全部保证金，但没有义务支付超出部分。投资者必须知晓并完全理解与市场和交易相关的所有风险。在交易任何产品之前，认真考虑自己的财务状况与经验水平。任何意见、消息、研究、分析、价格或者其他信息都只能作为一般市场评论，不包含任何投资建议。市场建议不是根据提升投资研究独立性的法律要求准备的，因此不受任何在传播之前交易的禁令限制。FXCM 和作者不承担损失的责任，包括并不限于任何因为依赖这些信息而造成的直接或间接利润损失。

FXCM 交易平台甚至允许资金头寸较小的个人交易者实施和部署算法交易策略。

本章介绍 FXCM 交易 API 的基本功能，以及通过编程实现自动算法交易策略所需的 fxcmpy Python 库。具体的组织结构如下。

入门

本节介绍如何准备 FXCM REST API 算法交易所需的一切条件。

读取数据

本节讲解读取及使用金融数据的方法（直到分笔数据级别）。

使用 API

本节阐述用 REST API 实现的典型任务，如读取历史及流数据、下单和查询账户信息。

14.1 入门

在命令行上执行如下命令来安装 Python 包装器库 fxcmpy：

```
pip install fxcmpy
```

fxcmpy 库的文档可以在 fxcmpy 官网上找到。对于 FXCM 交易 API 和 fxcmpy 库的入门学习，在 FXCM 上注册一个免费的演示账户就足够了[1]。下一步是在演示账户内创建唯一的 API 令牌——如 YOUR_FXCM_API_TOKEN。然后，打开到 API 的链接，例如通过如下语句：

```
import fxcmpy
api = fxcmpy.fxcmpy(access_token=YOUR_FXCM_API_TOKEN, log_level='error')
```

另外，也可以使用一个配置文件（如 fxcm.cfg）连接到 API。这个文件的内容如下：

```
[FXCM]
log_level = error
log_file = PATH_TO_AND_NAME_OF_LOG_FILE
access_token = YOUR_FXCM_API_TOKEN
```

然后，你可以通过以下语句连接到 API：

```
import fxcmpy
api = fxcmpy.fxcmpy(config_file='fxcm.cfg')
```

默认情况下，`fxcmpy` 类连接到演示服务器。不过，通过使用 `server` 参数，`fxcmpy` 类可以连接到真实的交易服务器（如果有账户的话）：

```
api = fxcmpy.fxcmpy(config_file='fxcm.cfg', server='demo') ❶
api = fxcmpy.fxcmpy(config_file='fxcm.cfg', server='real') ❷
```

❶ 连接到演示服务器。

1　注意，FXCM 演示账户仅在某些国家提供。——原注

❷ 连接到真实交易服务器。

14.2 读取数据

FXCM 提供对历史市场价格数据集（例如分笔交易数据）的访问，这些数据预先经过打包。例如，你可以从 FXCM 服务器读取包含 2018 年第 26 周欧元/美元汇率分时数据的压缩文件，这将在之后的内容中介绍。之后的内容还将解释从 API 读取历史 K 线（蜡烛图）数据的方法。

14.2.1 读取分笔交易数据

FXCM 提供多种货币组合的历史分笔交易数据。fxcmpy 库方便了这种分时数据的读取与使用。首先导入一些库：

```
In [1]: import time
        import numpy as np
        import pandas as pd
        import datetime as dt
        from pylab import mpl, plt

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

其次，查看有分时数据的产品代码（货币组合）：

```
In [3]: from fxcmpy import fxcmpy_tick_data_reader as tdr

In [4]: print(tdr.get_available_symbols())
        ('AUDCAD', 'AUDCHF', 'AUDJPY', 'AUDNZD', 'CADCHF', 'EURAUD', 'EURCHF',
        'EURGBP', 'EURJPY', 'EURUSD', 'GBPCHF', 'GBPJPY', 'GBPNZD', 'GBPUSD',
        'GBPCHF', 'GBPJPY', 'GBPNZD', 'NZDCAD', 'NZDCHF', 'NZDJPY', 'NZDUSD',
        'USDCAD', 'USDCHF', 'USDJPY')
```

下面的代码用于读取某个产品一周的分笔交易数据。得到的 pandas 的 `DataFrame` 对象有 150 万个数据行：

```
In [5]: start = dt.datetime(2018, 6, 25) ❶
        stop = dt.datetime(2018, 6, 30) ❶

In [6]: td = tdr('EURUSD', start, stop) ❶

In [7]: td.get_raw_data().info() ❷
        <class 'pandas.core.frame.DataFrame'>
        Index: 1963779 entries, 06/24/2018 21:00:12.290 to 06/29/2018
```

```
 20:59:00.607
Data columns (total 2 columns):
Bid    float64
Ask    float64
dtypes: float64(2)
memory usage: 44.9+ MB

In [8]: td.get_data().info() ❸
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 1963779 entries, 2018-06-24 21:00:12.290000 to
2018-06-29
         20:59:00.607000
        Data columns (total 2 columns):
        Bid    float64
        Ask    float64
        dtypes: float64(2)
        memory usage: 44.9 MB

In [9]: td.get_data().head()
Out[9]:                             Bid      Ask
        2018-06-24 21:00:12.290  1.1662  1.16660
        2018-06-24 21:00:16.046  1.1662  1.16650
        2018-06-24 21:00:22.846  1.1662  1.16658
        2018-06-24 21:00:22.907  1.1662  1.16660
        2018-06-24 21:00:23.441  1.1662  1.16663
```

❶ 读取数据文件、解压并将原始数据保存在 DataFrame 对象（作为结果对象的一个属性）中。

❷ td.get_raw_data()方法返回包含原始数据的 DataFrame 对象；索引值仍然是字符串对象。

❸ td.get_data()方法返回 DataFrame 对象，其索引已经转换为 DatetimeIndex。

由于分笔交易数据保存在 DataFrame 对象中，所以选择一个数据子集实施典型金融分析任务就很简单了。图 14-1 显示从该子集得到的中间价图表以及一条简单移动平均线（SMA）：

```
In [10]: sub = td.get_data(start='2018-06-29 12:00:00',
                           end='2018-06-29 12:15:00') ❶
In [11]: sub.head()
Out[11]:                             Bid      Ask
         2018-06-29 12:00:00.011  1.16497  1.16498
         2018-06-29 12:00:00.071  1.16497  1.16497
         2018-06-29 12:00:00.079  1.16497  1.16498
         2018-06-29 12:00:00.091  1.16495  1.16498
         2018-06-29 12:00:00.205  1.16496  1.16498
```

```
In [12]: sub['Mid'] = sub.mean(axis=1) ❷

In [13]: sub['SMA'] = sub['Mid'].rolling(1000).mean() ❸

In [14]: sub[['Mid', 'SMA']].plot(figsize=(10, 6), lw=0.75);
```

❶ 选择完整数据集的一个子集。

❷ 从买入价和卖出价计算中间价。

❸ 得出 1000 笔交易的 SMA 值。

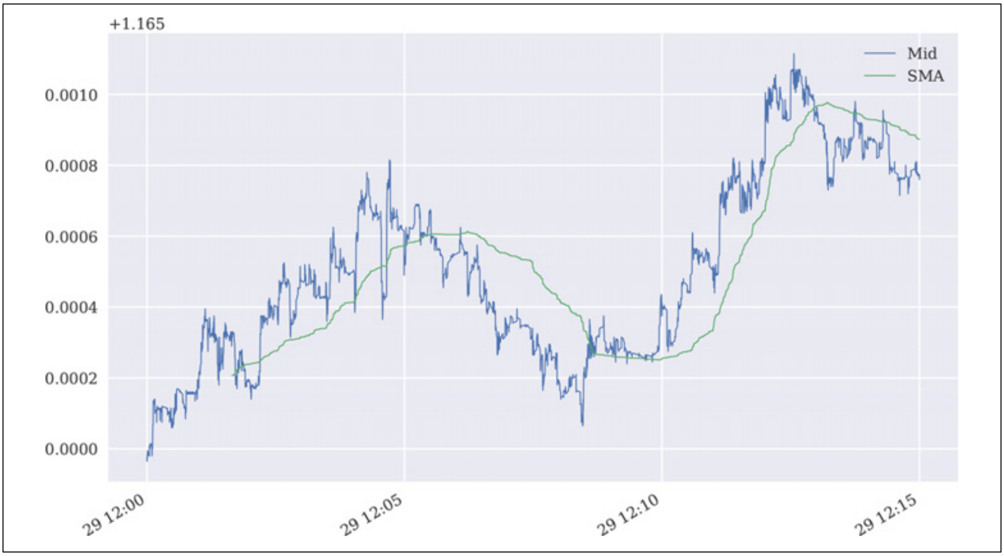


图 14-1　欧元/美元历史分笔中间价及 SMA

14.2.2　读取 K 线（蜡烛图）数据

FXCM 还提供了对历史 K 线（蜡烛图）数据的访问（在 API 之外），也就是访问某些均匀的时间间隔中的数据，包括买入与卖出的开盘价、最高价、最低价和收盘价。

首先查看提供蜡烛图数据的可用产品代码：

```
In [15]: from fxcmpy import fxcmpy_candles_data_reader as cdr

In [16]: print(cdr.get_available_symbols())
         ('AUDCAD', 'AUDCHF', 'AUDJPY', 'AUDNZD', 'CADCHF', 'EURAUD', 'EURCHF',
          'EURGBP', 'EURJPY', 'EURUSD', 'GBPCHF', 'GBPJPY', 'GBPNZD', 'GBPUSD',
          'GBPCHF', 'GBPJPY', 'GBPNZD', 'NZDCAD', 'NZDCHF', 'NZDJPY', 'NZDUSD',
          'USDCAD', 'USDCHF', 'USDJPY')
```

其次是读取数据，这与分笔数据读取类似。唯一的区别是需要指定 period 值——数据组长度（例如，ml 为一分钟，H1 为一小时，D1 为一天）：

```
In [17]: start = dt.datetime(2018, 5, 1)
         stop = dt.datetime(2018, 6, 30)

In [18]: period = 'H1' ❶

In [19]: candles = cdr('EURUSD', start, stop, period)
In [20]: data = candles.get_data()

In [21]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 1080 entries, 2018-04-29 21:00:00 to 2018-06-29
20:00:00
         Data columns (total 8 columns):
         BidOpen     1080 non-null float64
         BidHigh     1080 non-null float64
         BidLow      1080 non-null float64
         BidClose    1080 non-null float64
         AskOpen     1080 non-null float64
         AskHigh     1080 non-null float64
         AskLow      1080 non-null float64
         AskClose    1080 non-null float64
         dtypes: float64(8)
         memory usage: 75.9 KB

In [22]: data[data.columns[:4]].tail() ❷
Out[22]:                      BidOpen  BidHigh   BidLow  BidClose
         2018-06-29 16:00:00  1.16768  1.16820  1.16731   1.16769
         2018-06-29 17:00:00  1.16769  1.16826  1.16709   1.16781
         2018-06-29 18:00:00  1.16781  1.16816  1.16668   1.16684
         2018-06-29 19:00:00  1.16684  1.16792  1.16638   1.16774
         2018-06-29 20:00:00  1.16774  1.16904  1.16758   1.16816

In [23]: data[data.columns[4:]].tail() ❸
Out[23]:                      AskOpen  AskHigh   AskLow  AskClose
         2018-06-29 16:00:00  1.16769  1.16820  1.16732   1.16771
         2018-06-29 17:00:00  1.16771  1.16827  1.16711   1.16782
         2018-06-29 18:00:00  1.16782  1.16817  1.16669   1.16686
         2018-06-29 19:00:00  1.16686  1.16794  1.16640   1.16775
         2018-06-29 20:00:00  1.16775  1.16907  1.16760   1.16861
```

❶ 指定 period 值。

❷ 开盘、最高、最低、收盘买入价。

❸ 开盘、最高、最低、收盘卖出价。

本节的最后用以下代码来计算收盘中间价和两个 SMA，并绘制结果图表（见图 14-2）：

```
In [24]: data['MidClose'] = data[['BidClose', 'AskClose']].mean(axis=1) ❶

In [25]: data['SMA1'] = data['MidClose'].rolling(30).mean() ❷
         data['SMA2'] = data['MidClose'].rolling(100).mean() ❷

In [26]: data[['MidClose', 'SMA1', 'SMA2']].plot(figsize=(10, 6));
```

❶ 从收盘买入与卖出价计算出收盘中间价。

❷ 计算两条 SMA，一条用于较短的时间间隔，一条用于较长的时间间隔。



图 14-2　欧元/美元每小时历史收盘价及两条 SMA

14.3　使用 API

14.2 节介绍了 FXCM 服务器上预先打包的历史分笔数据和蜡烛图数据的读取方法，本节将介绍如何通过 API 读取历史数据。为此，我们需要一个 FXCM API 的连接对象。首先导入 fxcmpy 库，将其连接到 API（基于唯一的 API 令牌），并查看可用的金融工具：

```
In [27]: import fxcmpy

In [28]: fxcmpy.__version__
```

```
Out[28]: '1.1.33'

In [29]: api = fxcmpy.fxcmpy(config_file='../fxcm.cfg') ❶

In [30]: instruments = api.get_instruments()

In [31]: print(instruments)
         ['EUR/USD', 'XAU/USD', 'GBP/USD', 'UK100', 'USDOLLAR', 'XAG/
USD', 'GER30',
          'FRA40', 'USD/CNH', 'EUR/JPY', 'USD/JPY', 'CHN50', 'GBP/JPY',
'AUD/JPY',
          'CHF/JPY', 'USD/CHF', 'GBP/CHF', 'AUD/USD', 'EUR/AUD', 'EUR/CHF',
          'EUR/CAD', 'EUR/GBP', 'AUD/CAD', 'NZD/USD', 'USD/CAD', 'CAD/JPY',
          'GBP/AUD', 'NZD/JPY', 'US30', 'GBP/CAD', 'SOYF', 'GBP/NZD', 'AUD/NZD',
          'USD/SEK', 'EUR/SEK', 'EUR/NOK', 'USD/NOK', 'USD/MXN', 'AUD/CHF',
          'EUR/NZD', 'USD/ZAR', 'USD/HKD', 'ZAR/JPY', 'BTC/USD', 'USD/TRY',
          'EUR/TRY', 'NZD/CHF', 'CAD/CHF', 'NZD/CAD', 'TRY/JPY', 'AUS200',
          'ESP35', 'HKG33', 'JPN225', 'NAS100', 'SPX500', 'Copper', 'EUSTX50',
          'USOil', 'UKOil', 'NGAS', 'Bund']
```

❶ 连接到 API；调整路径/文件名。

14.3.1 读取历史数据

一旦连接，具体时段数据的读取通过一次方法调用即可实现。使用 `get_candles()` 方法时，`period` 参数可以是：`m1`、`m5`、`m15`、`m30`、`H1`、`H2`、`H3`、`H4`、`H6`、`H8`、`D1`、`W1` 或 `M1`。以下代码给出了几个例子。图 14-3 展示了欧元/美元工具（货币组合）收盘买入价的一分钟分时线：

```
In [32]: candles = api.get_candles('USD/JPY', period='D1', number=10) ❶

In [33]: candles[candles.columns[:4]] ❶
Out[33]:                      bidopen bidclose  bidhigh   bidlow
         date
         2018-10-08 21:00:00  113.760  113.219  113.937  112.816
         2018-10-09 21:00:00  113.219  112.946  113.386  112.863
         2018-10-10 21:00:00  112.946  112.267  113.281  112.239
         2018-10-11 21:00:00  112.267  112.155  112.528  111.825
         2018-10-12 21:00:00  112.155  112.200  112.491  111.873
         2018-10-14 21:00:00  112.163  112.130  112.270  112.109
         2018-10-15 21:00:00  112.130  111.758  112.230  111.619
         2018-10-16 21:00:00  112.151  112.238  112.333  111.727
         2018-10-17 21:00:00  112.238  112.636  112.670  112.009
         2018-10-18 21:00:00  112.636  112.168  112.725  111.942
```

```
In [34]: candles[candles.columns[4:]] ❶
Out[34]:                      askopen askclose askhigh  asklow  tickqty
         date
         2018-10-08 21:00:00 113.840  113.244 113.950 112.827   184835
         2018-10-09 21:00:00 113.244  112.970 113.399 112.875   321755
         2018-10-10 21:00:00 112.970  112.287 113.294 112.265   329174
         2018-10-11 21:00:00 112.287  112.175 112.541 111.835   568231
         2018-10-12 21:00:00 112.175  112.243 112.504 111.885   363233
         2018-10-14 21:00:00 112.219  112.181 112.294 112.145      581
         2018-10-15 21:00:00 112.181  111.781 112.243 111.631   322304
         2018-10-16 21:00:00 112.163  112.271 112.345 111.740   253420
         2018-10-17 21:00:00 112.271  112.664 112.682 112.022   542166
         2018-10-18 21:00:00 112.664  112.237 112.738 111.955   369012

In [35]: start = dt.datetime(2017, 1, 1) ❷
         end = dt.datetime(2018, 1, 1) ❷

In [36]: candles = api.get_candles('EUR/GBP', period='D1',
                                   start=start, stop=end) ❷

In [37]: candles.info() ❷
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 309 entries, 2017-01-03 22:00:00 to 2018-01-01 22:00:00
         Data columns (total 9 columns):
         bidopen     309 non-null float64
         bidclose    309 non-null float64
         bidhigh     309 non-null float64
         bidlow      309 non-null float64
         askopen     309 non-null float64
         askclose    309 non-null float64
         askhigh     309 non-null float64
         asklow      309 non-null float64
         tickqty     309 non-null int64
         dtypes: float64(8), int64(1)
         memory usage: 24.1 KB

In [38]: candles = api.get_candles('EUR/USD', period='m1', number=250) ❸

In [39]: candles['askclose'].plot(figsize=(10, 6))
```

❶ 读取最近的 10 个日终价格。

❷ 读取全年的日终价格。

❸ 读取最近可用的一分钟价格。

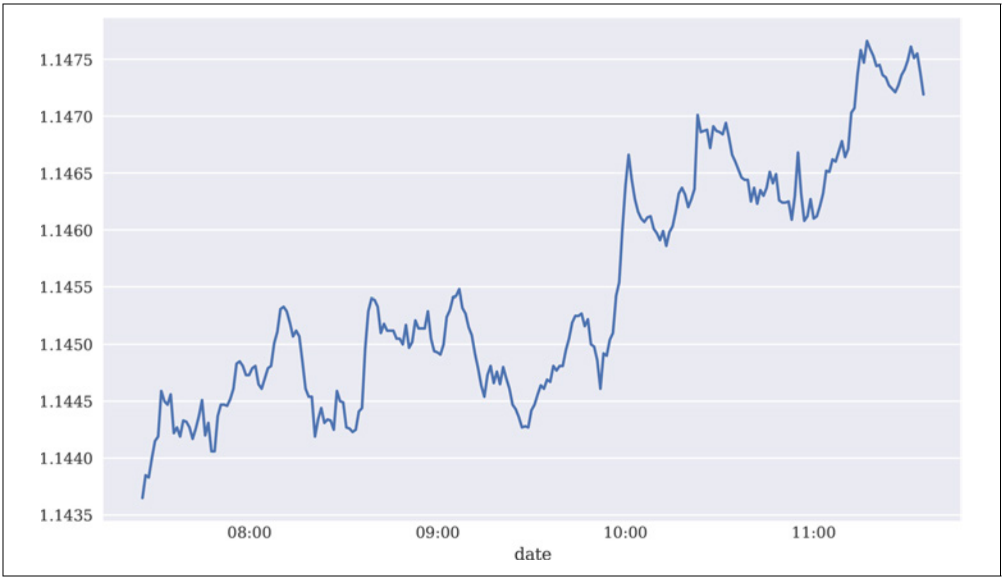


图 14-3　欧元/美元历史收盘买入价（分钟线）

14.3.2　读取流数据

虽然历史数据很重要（例如对算法交易策略的事后检验），但持续访问实时或者流数据（在交易时间内）是部署和自动化算法交易策略所必需的。FXCM API 可以订阅所有金融工具的实时数据流。fxcmpy 包装器库支持这一功能，允许用户提供自定义函数（所谓的“回调函数”）以处理实时数据流。

以下代码展示了一个简单的回调函数，它仅仅打印在读取数据集中选择的元素，并在订阅所需金融工具（这里是欧元/美元）之后用其实时处理数据：

```
In [40]: def output(data, dataframe):
             print('%3d | %s | %s | %6.5f, %6.5f'
                   % (len(dataframe), data['Symbol'],
                      pd.to_datetime(int(data['Updated']), unit='ms'),
                      data['Rates'][0], data['Rates'][1])) ❶

In [41]: api.subscribe_market_data('EUR/USD', (output,)) ❷
           1 | EUR/USD | 2018-10-19 11:36:39.735000 | 1.14694, 1.14705
           2 | EUR/USD | 2018-10-19 11:36:39.776000 | 1.14694, 1.14706
           3 | EUR/USD | 2018-10-19 11:36:40.714000 | 1.14695, 1.14707
           4 | EUR/USD | 2018-10-19 11:36:41.646000 | 1.14696, 1.14708
           5 | EUR/USD | 2018-10-19 11:36:41.992000 | 1.14696, 1.14709
           6 | EUR/USD | 2018-10-19 11:36:45.131000 | 1.14696, 1.14708
           7 | EUR/USD | 2018-10-19 11:36:45.247000 | 1.14696, 1.14709
```

```
In [42]: api.get_last_price('EUR/USD') ❸
Out[42]: Bid     1.14696
         Ask     1.14709
         High    1.14775
         Low     1.14323
         Name: 2018-10-19 11:36:45.247000, dtype: float64

In [43]: api.unsubscribe_market_data('EUR/USD') ❹
           8 | EUR/USD | 2018-10-19 11:36:48.239000 | 1.14696, 1.14708
```

❶ 可打印读取数据集中某些元素的回调函数。

❷ 订阅特定实时数据流；只要没有“解除订阅”事件，就异步处理数据。

❸ 订阅期间，`get_last_price()`方法返回最新的可用数据集。

❹ 解除实时数据流订阅。

> **回调函数**
>
> 回调函数是基于一个或者多个 Python 函数来处理实时流数据的灵活手段。它们可用于输入数据打印等简单任务，也可以用于根据在线交易算法（见第 16 章）生成买卖信号等复杂任务。

14.3.3　下单

FXCM API 可以进行 FXCM 交易应用中的各类交易（如进场单或移动止损单）并对其进行管理。但是，下面的代码只介绍基本的市场买卖单，一般来说，这对算法交易入门就已经足够了。我们首先确认没有开仓头寸，然后建立不同的头寸（通过`creat_market_buy_order()`方法）：

```
In [44]: api.get_open_positions() ❶
Out[44]: Empty DataFrame
         Columns: []
         Index: []

In [45]: order = api.create_market_buy_order('EUR/USD', 10) ❷

In [46]: sel = ['tradeId', 'amountK', 'currency',
                'grossPL', 'isBuy'] ❸

In [47]: api.get_open_positions()[sel] ❸
Out[47]:      tradeId  amountK currency grossPL isBuy
         0  132607899       10  EUR/USD 0.17436  True

In [48]: order = api.create_market_buy_order('EUR/GBP', 5) ❹
```

```
In [49]: api.get_open_positions()[sel]
Out[49]:      tradeId  amountK currency  grossPL  isBuy
         0  132607899       10  EUR/USD  0.17436   True
         1  132607928        5  EUR/GBP -1.53367   True
```

❶ 显示连接（默认）账户的开仓头寸。

❷ 在欧元/美元货币组合中建立 10 万个单位的头寸[1]。

❸ 仅显示开仓头寸中选择的要素。

❹ 在欧元/美元货币组合中再建立 5 万个单位的头寸。

`create_market_buy_order()`函数用于建立或者增加头寸，而 `create_market_sell_order()`函数用于平仓或者减少头寸。平仓还有更通用的方法，如以下代码所示：

```
In [50]: order = api.create_market_sell_order('EUR/USD', 3) ❶

In [51]: order = api.create_market_buy_order('EUR/GBP', 5) ❷

In [52]: api.get_open_positions()[sel] ❸
Out[52]:      tradeId  amountK currency  grossPL  isBuy
          0 132607899       10  EUR/USD  0.17436   True
          1 132607928        5  EUR/GBP -1.53367   True
          2 132607930        3  EUR/USD -1.33369  False
          3 132607932        5  EUR/GBP -1.64728   True

In [53]: api.close_all_for_symbol('EUR/GBP') ❹

In [54]: api.get_open_positions()[sel]
Out[54]:      tradeId  amountK currency  grossPL  isBuy
         0  132607899       10  EUR/USD  0.17436   True
         1  132607930        3  EUR/USD -1.33369  False

In [55]: api.close_all() ❺

In [56]: api.get_open_positions()
Out[56]: Empty DataFrame
         Columns: []
         Index: []
```

❶ 减少欧元/美元货币组合头寸。

❷ 增加欧元/英镑货币组合头寸。

1 对于货币组合金融工具来说，买卖数量为 1000 的倍数。还要注意，不同账户可能有不同的杠杆比率。这意味着根据相关杠杆比率，相同的头寸可能需要或多或少的保证金。在必要时将例子中的数量调整到较低值。——原注

❸ 欧元/英镑现在有两个多头头寸；与欧元/美元头寸相反，它们没有盈利。

❹ `close_all_for_symbol()`方法平掉指定代码的所有仓位。

❺ `close_all()`方法平掉所有开仓。

14.3.4 账户信息

除了建仓之外，FXCM API 也可以读取更多常规账户信息。例如，用户可以查询默认账户（如果有多个账户的话），或者净值及保证金情况：

```
In [57]: api.get_default_account() ❶
Out[57]: 1090495

In [58]: api.get_accounts().T ❷
Out[58]:                              0
         accountId            1090495
         accountName         01090495
         balance               4915.2
         dayPL                 -41.97
         equity                4915.2
         grossPL                    0
         hedging                    Y
         mc                         N
         mcDate
         ratePrecision              0
         t                          6
         usableMargin          4915.2
         usableMargin3         4915.2
         usableMargin3Perc        100
         usableMarginPerc         100
         usdMr                      0
         usdMr3                     0
```

❶ 显示默认 `accountId` 值。

❷ 显示所有账户的财务状况和一些参数。

14.4 结语

本章介绍了 FXCM 用于算法交易的 REST API，涵盖以下主题：

- API 使用前的准备；
- 读取历史分笔数据；
- 读取历史 K 线数据；

- 读取实时流数据；
- 下单买卖；
- 查询账户信息。

FXCM API 和 fxcmpy 包装器库当然还提供其他功能，但上述功能是开始算法交易所需的基本组件。

14.5 延伸阅读

关于 FXCM 交易 API 和 Python 包装器库的更多细节，请参考以下文档：

- Trading API；
- fxcmpy 库。

Python 算法交易的全面在线培训计划请参见 Python Quants 官网。

第 15 章

交易策略

他们真傻，以为可以从过去预知未来。

——《经济学家》杂志[1]

本章介绍的是算法交易策略的向量化事后检验。算法交易策略一词用于描述任何基于算法、在没有人工干预情况下自行决定金融产品多头、空投或中立头寸的金融交易策略。简单算法如“每隔五分钟对苹果公司股票交替建立多头和中立头寸”就满足上述定义。对于本章的目的，从技术上说，算法交易策略由一些 Python 代码表示，根据新数据决定买入或者卖出某个金融工具，以便建立多头、空头或者中立头寸。

本章不提供算法交易策略的概述（15.9 节列出了更详细的介绍算法交易策略的参考资料），而是将焦点放在精选的几个此类算法向量化事后检验的技术特征上。利用这种方法，通常可以将检验策略的金融数据当成一个总体来操纵，然后对保存金融数据的 NumPy 的 `ndarray` 和 pandas 的 `DataFrame` 对象应用向量化操作。[2]

本章的另一个重点是应用机器和深度学习算法来设计算法交易策略。为此，将分类算法在历史数据上进行训练，以预测未来的市场方向变动。这通常要求将实值金融数据变换为相对少量的分类值[3]，使我们能够利用此类算法的模式识别能力。

本章分为以下几部分：

1 来源：“Does the Past Predict the Future?”，Economist 官网，2009 年 9 月 23 日。——原注

2 另一种替代方法基于事件的事后检验，这种方法将在每个新数据点上显式循环，模拟市场新数据点的到来。——原注

3 注意，使用实值时，每种模式都可能是独特或者罕见的，难以训练一种算法并从观察到的模式中得出任何结论。——原注

简单移动平均数

本节的重点是基于简单移动平均数的算法交易策略，以及这种策略的事后检验。

随机游走假设

本节介绍随机游走假设。

线性 OLS 回归

本节研究使用 OLS 回归得出算法交易策略的方法。

聚类

本节探讨使用无监督学习算法得出算法交易策略的方法。

频率方法

本节介绍算法交易的一种简单频率论方法。

分类

本节将介绍用于算法交易的机器学习分类算法。

深度神经网络

本节重点介绍深度神经网络，以及如何将其用于算法交易。

15.1　简单移动平均数

基于简单移动平均数（SMA）的交易方法已经有数十年的历史（可以参见 Brock 等人的文章（1992））。虽然许多交易者在自主交易中使用 SMA，但也可以将其用于设计简单的算法交易策略。本章用 SMA 介绍算法交易策略的向量化事后检验，基础是第 8 章的技术分析示例。

15.1.1　数据导入

首先进行一些导入工作：

```
In [1]: import numpy as np
        import pandas as pd
        import datetime as dt
        from pylab import mpl, plt

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

其次，读取原始数据并选择苹果公司股票（AAPL.O）的金融时间序列。本节的分析基于日终数据；日内数据用于后续的几节。

```
In [3]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                          index_col=0, parse_dates=True)

In [4]: raw.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 2216 entries, 2010-01-01 to 2018-06-29
        Data columns (total 12 columns):
        AAPL.O    2138 non-null float64
        MSFT.O    2138 non-null float64
        INTC.O    2138 non-null float64
        AMZN.O    2138 non-null float64
        GS.N      2138 non-null float64
        SPY       2138 non-null float64
        .SPX      2138 non-null float64
        .VIX      2138 non-null float64
        EUR=      2216 non-null float64
        XAU=      2211 non-null float64
        GDX       2138 non-null float64
        GLD       2138 non-null float64
        dtypes: float64(12)
        memory usage: 225.1 KB

In [5]: symbol = 'AAPL.O'

In [6]: data = (
            pd.DataFrame(raw[symbol])
            .dropna()
        )
```

15.1.2 交易策略

接下来，为两种不同的滚动窗口大小计算 SMA 值。图 15-1 直观地展示了这 3 个时间序列：

```
In [7]: SMA1 = 42 ❶
        SMA2 = 252 ❷

In [8]: data['SMA1'] = data[symbol].rolling(SMA1).mean() ❶
        data['SMA2'] = data[symbol].rolling(SMA2).mean() ❷

In [9]: data.plot(figsize=(10, 6));
```

❶ 计算短期 SMA 值。

❷ 计算长期 SMA 值。



图 15-1　苹果股价与两条简单移动平均线

最后得出头寸。交易规则是：

- 短期 SMA 在长期 SMA 之上时买入（=+1）；
- 短期 SMA 在长期 SMA 之下时卖出（=-1）[1]。

头寸的可视化结果如图 15-2 所示：

```
In [10]: data.dropna(inplace=True)

In [11]: data['Position'] = np.where(data['SMA1'] > data['SMA2'], 1, -1) ❶

In [12]: data.tail()
Out[12]:             AAPL.O        SMA1        SMA2  Position
         Date
         2018-06-25  182.17  185.606190  168.265556         1
         2018-06-26  184.43  186.087381  168.418770         1
         2018-06-27  184.16  186.607381  168.579206         1
         2018-06-28  185.50  187.089286  168.736627         1
         2018-06-29  185.11  187.470476  168.901032         1

In [13]: ax = data.plot(secondary_y='Position', figsize=(10, 6))
         ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```

1　类似地，对于仅多头策略，将以 1 代表多头头寸，0 代表中立头寸。——原注

❶ np.where(cond, a, b)逐个元素地计算条件 cond，如果为 True 则设值 a，否则设值 b。



图 15-2 苹果公司股价、两条 SMA 以及结果头寸

以上复制了第 8 章得出的结果。但当时没有说明，遵循交易规则操作也就是实施算法交易策略能否得到比基准（在整段期间里看多苹果股票）更好的结果。鉴于这种策略只在两段时间里指示卖出苹果股票，所以只有这两段时期产生了绩效上的差别。

15.1.3 向量化事后检验

现在，向量化事后检验可以按照以下方式实施。首先计算对数收益率。然后将头寸（以 +1 或者−1 表示）乘以对应的对数回报率。这种简单的计算是可能的，因为多头头寸赚取的是苹果股票的正收益，而空头头寸赚取的是苹果股票的负收益。最终，将苹果公司股票的对数收益率与基于 SMA 的算法交易策略加总，并对其应用指数函数以得出绩效：

```
In [14]: data['Returns'] = np.log(data[symbol] / data[symbol].shift(1)) ❶

In [15]: data['Strategy'] = data['Position'].shift(1) * data['Returns'] ❷

In [16]: data.round(4).head()
Out[16]:             AAPL.O     SMA1     SMA2  Position  Returns  Strategy
         Date
         2010-12-31  46.0800  45.2810  37.1207         1      NaN       NaN
         2011-01-03  47.0814  45.3497  37.1862         1   0.0215    0.0215
         2011-01-04  47.3271  45.4126  37.2525         1   0.0052    0.0052
         2011-01-05  47.7142  45.4661  37.3223         1   0.0081    0.0081
         2011-01-06  47.6757  45.5226  37.3921         1  -0.0008   -0.0008
```

```
In [17]: data.dropna(inplace=True)

In [18]: np.exp(data[['Returns', 'Strategy']].sum()) ❸
Out[18]: Returns     4.017148
         Strategy    5.811299
         dtype: float64

In [19]: data[['Returns', 'Strategy']].std() * 252 ** 0.5 ❹
Out[19]: Returns     0.250571
         Strategy    0.250407
         dtype: float64
```

❶ 计算评估公司股票的对数收益率（即基准投资）。

❷ 将头寸值（移位一天）乘以苹果公司的对数收益率；移位是避免后见偏差所必需的[1]。

❸ 加总策略的对数回报率和基准投资，计算其指数值以得出绝对绩效。

❹ 计算策略与基准投资的年化波动率。

这些数字说明，算法交易策略确实胜过被动持有苹果股票的基准投资。由于策略的类型和特性以及年化波动率相同，所以策略本身也胜过风险调整基础上的基准投资。

为了更好地理解总体绩效，图 15-3 展示了苹果公司股票及算法交易策略在一段时间里的表现：



图 15-3　苹果公司股票及基于 SAM 的交易策略在一段时间内的表现

1　基本思路是，这种算法只能按照当天的市场数据（例如，收盘前一刻）建立苹果股票头寸。然后，这一头寸赚取的是明天的收益。——原注

```
In [20]: ax = data[['Returns', 'Strategy']].cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6))
         data['Position'].plot(ax=ax, secondary_y='Position', style='--')
         ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```

简化

本节介绍的向量化事后检验方法基于一些简化假设。交易成本（固定费用、买卖价差、贷款成本等）没有包含在内。对于在多年内只导致少数交易的交易策略来说，这可能是合理的。更为真实的事后检验方法应该考虑这些因素和其他（市场微观结构）因素。

15.1.4　优化

人们自然会问，选择参数 SMA1=42 和 SMA2=252 是否“正确”。一般来说，投资者希望在其他条件相同的情况下得到更高的收益。因此，他们可能倾向于寻找最大化对应时期内收益的参数。为此，可以使用暴力方法，对不同参数组合并重复整个向量化事后检验过程，记录结果，然后进行排名。下面的代码就用于实现这个功能：

```
In [21]: from itertools import product

In [22]: sma1 = range(20, 61, 4) ❶
         sma2 = range(180, 281, 10) ❷

In [23]: results = pd.DataFrame()
         for SMA1, SMA2 in product(sma1, sma2): ❸
             data = pd.DataFrame(raw[symbol])
             data.dropna(inplace=True)
             data['Returns'] = np.log(data[symbol] / data[symbol].shift(1))
             data['SMA1'] = data[symbol].rolling(SMA1).mean()
             data['SMA2'] = data[symbol].rolling(SMA2).mean()
             data.dropna(inplace=True)
             data['Position'] = np.where(data['SMA1'] > data['SMA2'], 1, -1)
             data['Strategy'] = data['Position'].shift(1) * data['Returns']
             data.dropna(inplace=True)
             perf = np.exp(data[['Returns', 'Strategy']].sum())
             results = results.append(pd.DataFrame(
                         {'SMA1': SMA1, 'SMA2': SMA2,
                          'MARKET': perf['Returns'],
                          'STRATEGY': perf['Strategy'],
                          'OUT': perf['Strategy'] - perf['Returns']},
                          index=[0]), ignore_index=True) ❹
```

❶ 指定 SMA1 参数值。

❷ 指定 SMA2 参数值。

❸ 组合所有 SMA1 和 SMA2 值。

❹ 将向量化事后检验结果记录在一个 DataFrame 对象。

下面的代码给出结果的概要，并显示事后检验中绩效最佳的 7 个参数组合。排名的根据是算法交易策略绩效与基准投资相比胜出的比例。由于 SMA2 参数的选择影响时间间隔长度和实施向量化事后检验的数据集，因此基准投资的绩效也各不相同：

```
In [24]: results.info()
         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 121 entries, 0 to 120
         Data columns (total 5 columns):
         SMA1        121 non-null int64
         SMA2        121 non-null int64
         MARKET      121 non-null float64
         STRATEGY    121 non-null float64
         OUT         121 non-null float64
         dtypes: float64(3), int64(2)
         memory usage: 4.8 KB

In [25]: results.sort_values('OUT', ascending=False).head(7)
Out[25]:      SMA1  SMA2    MARKET  STRATEGY       OUT
         56     40   190  4.650342  7.175173  2.524831
         39     32   240  4.045619  6.558690  2.513071
         59     40   220  4.220272  6.544266  2.323994
         46     36   200  4.074753  6.389627  2.314874
         55     40   180  4.574979  6.857989  2.283010
         70     44   220  4.220272  6.469843  2.249571
         101    56   200  4.074753  6.319524  2.244772
```

根据基于暴力法的优化结果，SMA1=40 和 SMA2=190 是最优参数，绩效比基准高出 230%。但是，这一结果严重依赖使用的数据集，因此很容易导致过度拟合。更为严格的方法是在一个数据集（样本内或者训练数据集）上实施优化，并在另一个数据集（样本外或测试数据集）上检验。

过度拟合

一般来说，算法交易策略背景下任何类型的优化、拟合或者训练都很容易导致过度拟合。这意味着，选择的参数可能对于使用的数据集表现（异常）良好，但在其他数据集或者实践中却表现（异常）糟糕。

15.2 随机游走假设

前一节介绍了向量化事后检验，它是算法交易策略事后检验的有效工具。我们检验的策略基于单一金融时间序列——苹果公司股票的历史日终价格，该策略的绩效好于同期简单看多苹果公司股票的基准投资。

虽然本质上很清楚，但这些结果与随机游走假设（RWH）的预测相反。根据这种假设，此类预测性方法完全不应该得到任何效益上的增长。RWH 假设金融市场中的价格遵循随机游走模式，或者在连续的时间里呈无漂移的算术布朗运动。未来任何时间点，无漂移算术布朗运动的预期值等于今天的值。[1]因此，从最小二乘法的意义上，如果 RWH 成立，对明日价格的最佳预测就是今日的价格。

以下引语总结了这些结果：

> 多年以来，经济学家、统计学家和金融教师都对开发和检验股票价格表现模型很感兴趣。随机游走理论就是从这种研究中演化出来的重要模型之一。这一理论对其他许多描述与预测股价表现的理论持严重怀疑的态度，尽管后者在学术界之外相当流行。例如，我们以后将会看到，如果随机游走是对现实的精确描述，那么各种预测股价的“技术”或者“图表”方法都将变得毫无价值。
>
> ——Eugene F. Fama (1965)

RWH 与有效市场假设（EMH）一致，用非技术性的话说，后者假设市场价格反映了“所有可用信息”。通常要区分不同的市场有效度，如“弱”“半强”和“强”，可以更加明确地定义“所有可用信息”的涵盖范围。从形式上说，这样的定义可以基于理论上的信息集概念和用于编程目的的数据集，正如下面的引语所述：

> 如果根据信息集 S 交易无法得到经济利益，那么市场对该信息集来说就是有效的。
>
> ——Michael Jensen (1978)

下面，我们可以使用 Python 来对特定案例检验 RWH。我们将使用一个历史市场价格的金融时间序列，并为其创建一些（如 5 个）滞后版本。然后，使用 OLS 回归，根据之前创建的滞后市场价格预测市场价格。基本思路是，昨天以及之前 4 天的市场价格可用于预测今天的市场价格。

下面的 Python 代码可以实现上述思路，创建标普 500 股票指数历史日终收盘价的 5 个滞后版本：

```
In [26]: symbol = '.SPX'

In [27]: data = pd.DataFrame(raw[symbol])
```

1 随机游走及基于布朗运动过程的正式定义及更详细的讨论参见 Baxter 和 Rennie 的著作（1996）。——原注

```
In [28]: lags = 5
         cols = []
         for lag in range(1, lags + 1):
             col = 'lag_{}'.format(lag) ❶
             data[col] = data[symbol].shift(lag) ❷
             cols.append(col) ❸

In [29]: data.head(7)
Out[29]:                .SPX    lag_1    lag_2    lag_3    lag_4    lag_5
         Date
         2010-01-01      NaN      NaN      NaN      NaN      NaN      NaN
         2010-01-04  1132.99      NaN      NaN      NaN      NaN      NaN
         2010-01-05  1136.52  1132.99      NaN      NaN      NaN      NaN
         2010-01-06  1137.14  1136.52  1132.99      NaN      NaN      NaN
         2010-01-07  1141.69  1137.14  1136.52  1132.99      NaN      NaN
         2010-01-08  1144.98  1141.69  1137.14  1136.52  1132.99      NaN
         2010-01-11  1146.98  1144.98  1141.69  1137.14  1136.52  1132.99

In [30]: data.dropna(inplace=True)
```

❶ 为当前 `lag` 值定义列名。

❷ 按照当前 `lag` 值创建市场价格的滞后版本。

❸ 收集后续参考点的列名。

使用 NumPy 很容易实现 OLS 回归。正如最优回归参数所示，`lag_1` 确实对基于 OLS 回归的市场价格预测非常重要。它的值接近于 1。其他 4 个值接近 0。图 15-4 可视化了最优回归参数值。

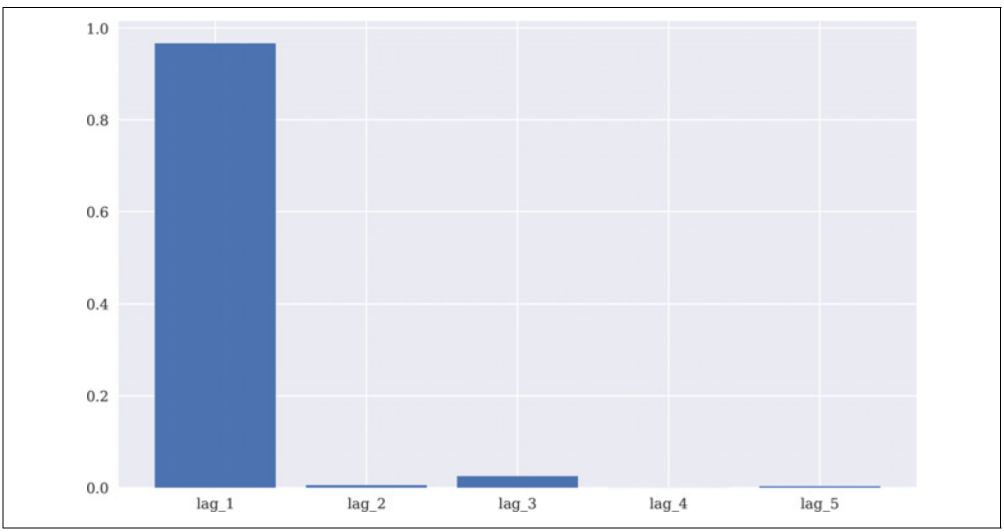


图 15-4　OLS 回归价格预测的最优回归参数

图 15-5 使用最优化结果可视化预测值，将其与原始的标普 500 指数值对比。从图 15-5 中可以明显看出，预测值本质上是利用 `lag_1` 提出的。图 15-5 中的预测线就是原始时间序列向右移动一天的结果（有一些小调整）。

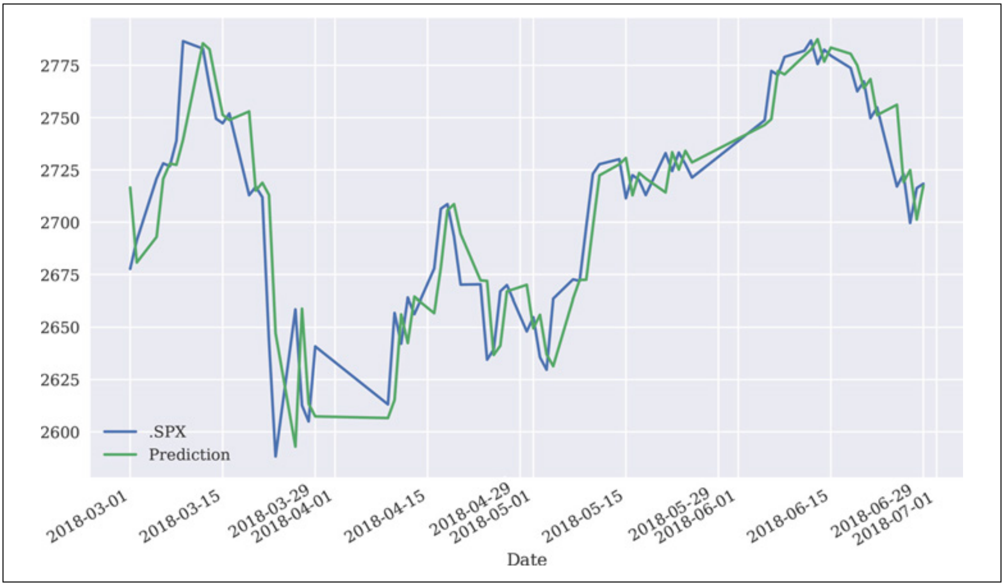


图 15-5　标普 500 指数水平与 OLS 回归预测值的对比

总而言之，本节的简短分析揭示了对 RWH 和 EMH 的某些支持。当然，这一分析仅针对一种股票指数。该分析还使用相当特殊的参数——但很容易扩展以加入不同类型资产的多种金融工具、不同的滞后值等。一般来说，从定性角度看，结果多少有些相似。毕竟，RWH 和 EMH 是得到广泛实证支持的金融理论。从这个意义上讲，任何算法交易策略都必须证明 RWH 不普遍适用，才能证明自身的价值。这当然是一个巨大的障碍。

15.3　线性 OLS 回归

本节应用线性 OLS 回归来根据历史对数收益率预测市场运动方向。简单起见，我们只使用两个特征。第一个特征（`lag_1`）表示滞后一天的金融时间序列对数收益率。第二个特征（`lag_2`）的对数收益率为滞后两天的。与价格不同，对数收益率通常是固定的，这常常是应用统计学和 ML 算法的必要条件。

使用滞后对数收益率作为特征的基本思路是，它们可以为预测未来收益提供信息。例如，人们可能假设两次向下运动之后，更有可能出现一次向上运动（平均值活动），或

者正相反，更有可能出现又一次向下运动（“势头”或者“趋势”）。回归技术的应用可以规范化这些非正式推理。

15.3.1　数据

首先导入并准备数据集。图 15-6 展示了欧元/美元汇率每日历史对数收益率的频率分布。它们是后面使用的特征及标签的基础：

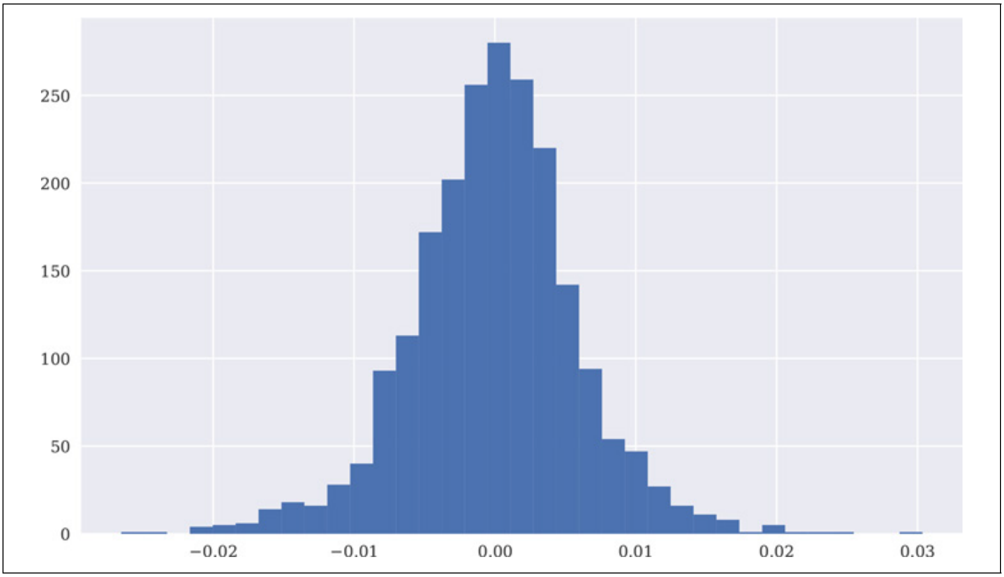


图 15-6　欧元/美元汇率对数收益率直方图

```
In [3]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                          index_col=0, parse_dates=True).dropna()

In [4]: raw.columns
Out[4]: Index(['AAPL.O', 'MSFT.O', 'INTC.O', 'AMZN.O', 'GS.N', 'SPY', '.SPX',
               '.VIX', 'EUR=', 'XAU=', 'GDX', 'GLD'],
               dtype='object')

In [5]: symbol = 'EUR='

In [6]: data = pd.DataFrame(raw[symbol])

In [7]: data['returns'] = np.log(data / data.shift(1))

In [8]: data.dropna(inplace=True)
```

```
In [9]: data['direction'] = np.sign(data['returns']).astype(int)

In [10]: data.head()
Out[10]:               EUR=   returns  direction
         Date
         2010-01-05  1.4368 -0.002988         -1
         2010-01-06  1.4412  0.003058          1
         2010-01-07  1.4318 -0.006544         -1
         2010-01-08  1.4412  0.006544          1
         2010-01-11  1.4513  0.006984          1

In [11]: data['returns'].hist(bins=35, figsize=(10, 6));
```

其次是将对数收益率滞后一天以创建特征，并将其与收益率数据组合显示（见图 15-7）：

```
In [12]: lags = 2

In [13]: def create_lags(data):
             global cols
             cols = []
             for lag in range(1, lags + 1):
                 col = 'lag_{}'.format(lag)
                 data[col] = data['returns'].shift(lag)
                 cols.append(col)

In [14]: create_lags(data)

In [15]: data.head()
Out[15]:               EUR=   returns  direction     lag_1     lag_2
         Date
         2010-01-05  1.4368 -0.002988         -1       NaN       NaN
         2010-01-06  1.4412  0.003058          1 -0.002988       NaN
         2010-01-07  1.4318 -0.006544         -1  0.003058 -0.002988
         2010-01-08  1.4412  0.006544          1 -0.006544  0.003058
         2010-01-11  1.4513  0.006984          1  0.006544 -0.006544

In [16]: data.dropna(inplace=True)

In [17]: data.plot.scatter(x='lag_1', y='lag_2', c='returns',
                           cmap='coolwarm', figsize=(10, 6), colorbar=True)
         plt.axvline(0, c='r', ls='--')
         plt.axhline(0, c='r', ls='--');
```

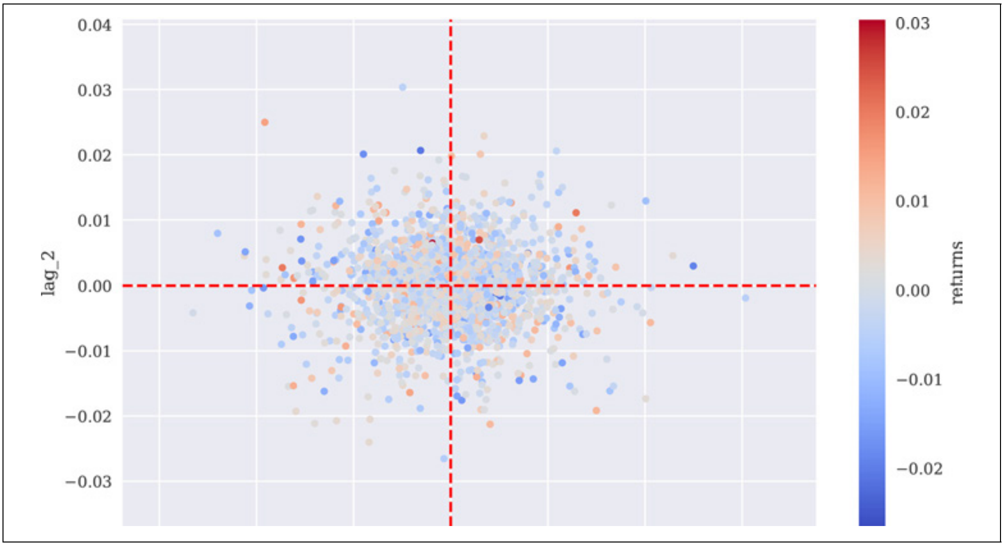


图 15-7　基于特征和标签数据的散点图

15.3.2　回归

数据集完成后，可以应用线性 OLS 回归学习任何潜在（线性）关系，然后根据特征预测市场行情，并根据预测检验交易策略。有两种基本方法用于预测：使用对数收益率或者仅用方向数据作为回归因变量。无论如何，预测结果是个实值，可将其变换成为 +1 或者−1 以仅用于方向的预测：

```
In [18]: from sklearn.linear_model import LinearRegression ❶

In [19]: model = LinearRegression() ❶

In [20]: data['pos_ols_1'] = model.fit(data[cols],
                                       data['returns']).predict(data
[cols]) ❷

In [21]: data['pos_ols_2'] = model.fit(data[cols],
                                       data['direction']).predict(data
[cols]) ❸

In [22]: data[['pos_ols_1', 'pos_ols_2']].head()
Out[22]:             pos_ols_1  pos_ols_2
         Date
         2010-01-07  -0.000166  -0.000086
         2010-01-08   0.000017   0.040404
         2010-01-11  -0.000244  -0.011756
         2010-01-12  -0.000139  -0.043398
```

```
        2010-01-13  -0.000022   0.002237

In [23]: data[['pos_ols_1', 'pos_ols_2']] = np.where(
                     data[['pos_ols_1', 'pos_ols_2']] > 0, 1, -1) ❹

In [24]: data['pos_ols_1'].value_counts() ❺
Out[24]: -1    1847
          1     288
         Name: pos_ols_1, dtype: int64

In [25]: data['pos_ols_2'].value_counts() ❺
Out[25]:  1   1377
         -1    758
         Name: pos_ols_2, dtype: int64

In [26]: (data['pos_ols_1'].diff() != 0).sum() ❻
Out[26]: 555

In [27]: (data['pos_ols_2'].diff() != 0).sum() ❻
Out[27]: 762
```

❶ 使用来自 scikit-learn 的线性 OLS 回归实现。

❷ 直接在对数收益率上实施回归。

❸ 也在最感兴趣的方向数据上进行。

❹ 市值预测转换为方向值（+1，−1）。

❺ 两种方法通常得到不同的方向预测值。

❻ 不过，两种方法都导致一段时间里相对多的交易。

有了方向预测，我们就可以应用向量化事后检验来判断所得交易策略的绩效。在这一阶段，分析的基础是一些简化的假设，例如“零交易成本”或使用同一数据集进行训练和测试。不过，在这些假设下，基于回归的策略表现都好于基准被动投资，但只有根据市场方向训练的策略整体收益为正数（见图 15-8）：

```
In [28]: data['strat_ols_1'] = data['pos_ols_1'] * data['returns']

In [29]: data['strat_ols_2'] = data['pos_ols_2'] * data['returns']

In [30]: data[['returns', 'strat_ols_1', 'strat_ols_2']].sum().apply(np.exp)
Out[30]: returns       0.810644
         strat_ols_1  0.942422
         strat_ols_2  1.339286
         dtype: float64
```

```
In [31]: (data['direction'] == data['pos_ols_1']).value_counts() ❶
Out[31]: False    1093
         True     1042
         dtype: int64

In [32]: (data['direction'] == data['pos_ols_2']).value_counts() ❶
Out[32]: True     1096
         False    1039
         dtype: int64

In [33]: data[['returns', 'strat_ols_1', 'strat_ols_2']].cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6));
```

❶ 显示策略的正确与错误预测数。



图 15-8　欧元/美元和基于回归的策略在一段时间里的表现

15.4　聚类

本节将 13.4 节介绍的 *k* 均值聚类应用到金融时间序列数据上，自动得到用于制定交易策略的聚类。具体思路是，通过算法确定两个特征值聚类来预测向上或向下运动。

以下代码对前面使用的两个特征应用 *k* 均值算法。图 15-9 是这两个聚类可视化的结果：

```
In [34]: from sklearn.cluster import KMeans

In [35]: model = KMeans(n_clusters=2, random_state=0) ❶
```

```
In [36]: model.fit(data[cols])
Out[36]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
             n_clusters=2, n_init=10, n_jobs=None, precompute_distances='auto',
             random_state=0, tol=0.0001, verbose=0)

In [37]: data['pos_clus'] = model.predict(data[cols])

In [38]: data['pos_clus'] = np.where(data['pos_clus'] == 1, -1, 1) ❷

In [39]: data['pos_clus'].values
Out[39]: array([-1, 1, -1, ..., 1, 1, -1])

In [40]: plt.figure(figsize=(10, 6))
         plt.scatter(data[cols].iloc[:, 0], data[cols].iloc[:, 1],
                     c=data['pos_clus'], cmap='coolwarm');
```

❶ 为算法选择的两个聚类。

❷ 根据聚类值选择头寸。

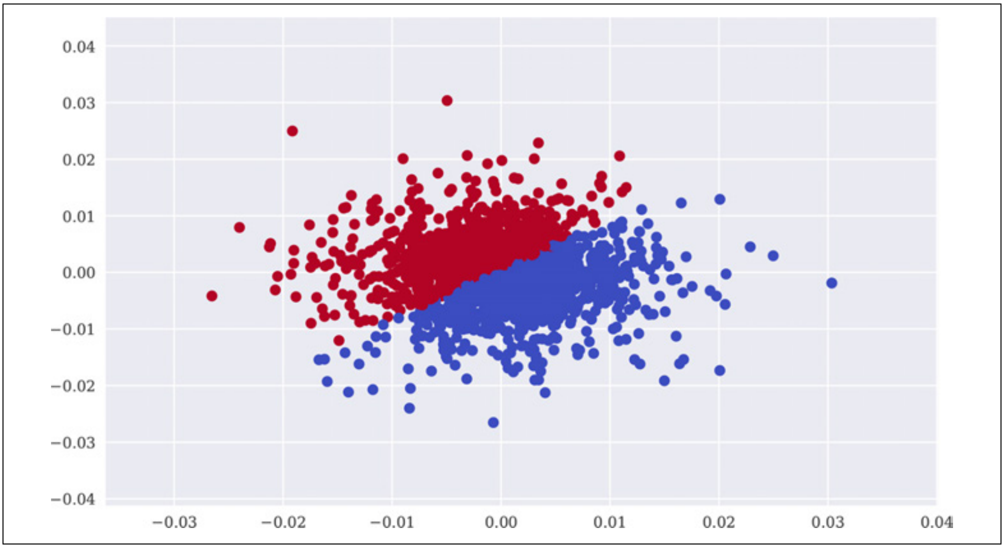


图 15-9　*k* 均值算法识别的两个聚类

必须承认，这种方法在此时相当武断——毕竟，算法如何知道人们的期望？但是，得到的交易策略最终略胜于基准被动投资（见图 15-10）。值得一提的是，我们没有进行任何指导（监督），命中率（正确预测次数与所有预测次数的比值）低于 50%：

```
In [41]: data['strat_clus'] = data['pos_clus'] * data['returns']

In [42]: data[['returns', 'strat_clus']].sum().apply(np.exp)
```

```
Out[42]: returns       0.810644
         strat_clus    1.277133
         dtype: float64

In [43]: (data['direction'] == data['pos_clus']).value_counts()
Out[43]: True      1077
         False     1058
         dtype: int64

In [44]: data[['returns', 'strat_clus']].cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6));
```



图 15-10　欧元/美元与基于 *k* 均值的策略在一段时间内的表现

15.5　频率方法

除了更复杂、精密的算法和技术之外，人们还可能提出采用频率方法以预测金融市场方向变动的思路。为此，可以将两个实值特征变换为二元特征。再根据两个二元特征的 4 种可能组合（(0, 0), (0, 1), (1, 0), (1, 1)），从向上和向下运动的历史观测数据中分别预测两种趋势的概率。

利用 pandas 的数据分析能力，这样的方法相对容易实现：

```
In [45]: def create_bins(data, bins=[0]):
             global cols_bin
             cols_bin = []
```

```
            for col in cols:
                col_bin = col + '_bin'
                data[col_bin] = np.digitize(data[col], bins=bins) ❶
                cols_bin.append(col_bin)

In [46]: create_bins(data)

In [47]: data[cols_bin + ['direction']].head() ❷
Out[47]:             lag_1_bin  lag_2_bin  direction
         Date
         2010-01-07          1          0         -1
         2010-01-08          0          1          1
         2010-01-11          1          0          1
         2010-01-12          1          1         -1
         2010-01-13          0          1          1

In [48]: grouped = data.groupby(cols_bin + ['direction'])
         grouped.size() ❸
Out[48]: lag_1_bin  lag_2_bin  direction
         0          0          -1           239
                                0             4
                                1           258
                    1          -1           262
                                1           288
         1          0          -1           272
                                0             1
                                1           278
                    1          -1           278
                                0             4
                                1           251
         dtype: int64

In [49]: res = grouped['direction'].size().unstack(fill_value=0) ❹

In [50]: def highlight_max(s):
             is_max = s == s.max()
             return ['background-color: yellow' if v else '' for v
in is_max] ❺

In [51]: res.style.apply(highlight_max, axis=1) ❺
Out[51]: <pandas.io.formats.style.Styler at 0x1a194216a0>
```

❶ 按照 bins 参数来数字化特征值。

❷ 显示数字化之后的特征值和标签值。

❸ 按照特征值组合，显示可能的运动情况的频率。

❹ 转换 `DataFrame` 对象，在各列中填入频率。

❺ 高亮显示每个特征值组合的最高频率值。

根据频率数据，3 个特征值组合表明向下运动，另一个似乎更可能表示向上运动。将此转换成一个交易策略，其结果如图 15-11 所示：

```
In [52]: data['pos_freq'] = np.where(data[cols_bin].sum(axis=1) == 2, -1, 1) ❶

In [53]: (data['direction'] == data['pos_freq']).value_counts()
Out[53]: True     1102
         False    1033
         dtype: int64

In [54]: data['strat_freq'] = data['pos_freq'] * data['returns']

In [55]: data[['returns', 'strat_freq']].sum().apply(np.exp)
Out[55]: returns       0.810644
         strat_freq    0.989513
         dtype: float64

In [56]: data[['returns', 'strat_freq']].cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6));
```

❶ 将频率上的发现转换成一个交易策略。

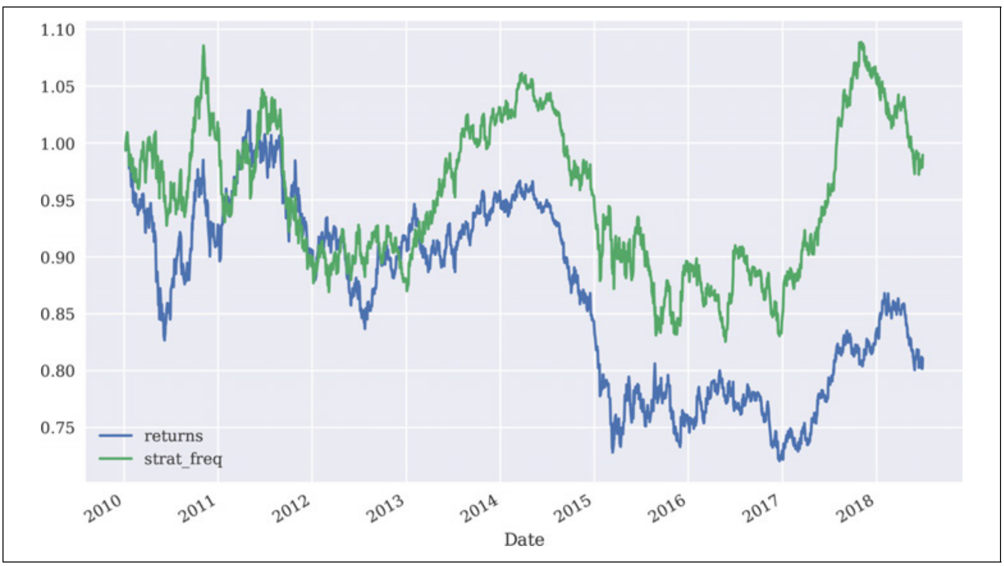


图 15-11　欧元/美元与基于频率的交易策略在一段时间里的表现

15.6 分类

本节将 ML 的分类算法（在 13.4 节中介绍过）应用到预测金融市场价格变动方向的问题上。按照前一节的背景知识和示例，逻辑回归、高斯朴素贝叶斯和支持向量机的应用与在较小样本数据集上的应用一样简单。

15.6.1 两个二元特征

首先是基于二元特征值的模型拟合，得出结果头寸值：

```
In [57]: from sklearn import linear_model
         from sklearn.naive_bayes import GaussianNB
         from sklearn.svm import SVC

In [58]: C = 1

In [59]: models = {
             'log_reg': linear_model.LogisticRegression(C=C),
             'gauss_nb': GaussianNB(),
             'svm': SVC(C=C)
         }

In [60]: def fit_models(data): ❶
         mfit = {model: models[model].fit(data[cols_bin],
                                          data['direction'])
                 for model in models.keys()}

In [61]: fit_models(data)

In [62]: def derive_positions(data): ❷
             for model in models.keys():
                 data['pos_' + model] = models[model].predict(data[cols_bin])

In [63]: derive_positions(data)
```

❶ 拟合所有模型的函数。

❷ 从拟合模型中得出所有头寸值的函数。

其次，对得出的交易策略进行向量化事后检验。图 15-12 可视化了一段时间的结果：

```
In [64]: def evaluate_strats(data): ❶
             global sel
             sel = []
             for model in models.keys():
                 col = 'strat_' + model
```

```
            data[col] = data['pos_' + model] * data['returns']
            sel.append(col)
        sel.insert(0, 'returns')

In [65]: evaluate_strats(data)

In [66]: sel.insert(1, 'strat_freq')

In [67]: data[sel].sum().apply(np.exp) ❷
Out[67]: returns           0.810644
         strat_freq        0.989513
         strat_log_reg     1.243322
         strat_gauss_nb    1.243322
         strat_svm         0.989513
         dtype: float64

In [68]: data[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 求得所有结果交易策略的函数。

❷ 一些策略可能有完全相同的表现。



图 15-12　欧元/美元与基于分类的交易策略（两个二元滞后特征）的表现

15.6.2　5 个二元特征

为了改善策略的绩效，下面的代码使用 5 个二元滞后特征来代替原来的两个特征。特别是，基于 SVM 的策略表现明显提高（见图 15-13）。而基于 LR 和 GNB 的绩效变

得更差：

```
In [69]: data = pd.DataFrame(raw[symbol])

In [70]: data['returns'] = np.log(data / data.shift(1))

In [71]: data['direction'] = np.sign(data['returns'])

In [72]: lags = 5 ❶
         create_lags(data)
         data.dropna(inplace=True)

In [73]: create_bins(data) ❷
         cols_bin
Out[73]: ['lag_1_bin', 'lag_2_bin', 'lag_3_bin', 'lag_4_bin', 'lag_5_bin']

In [74]: data[cols_bin].head()
Out[74]:             lag_1_bin  lag_2_bin  lag_3_bin  lag_4_bin  lag_5_bin
         Date
         2010-01-12          1          1          0          1          0
         2010-01-13          0          1          1          0          1
         2010-01-14          1          0          1          1          0
         2010-01-15          0          1          0          1          1
         2010-01-19          0          0          1          0          1

In [75]: data.dropna(inplace=True)

In [76]: fit_models(data)

In [77]: derive_positions(data)

In [78]: evaluate_strats(data)

In [79]: data[sel].sum().apply(np.exp)
Out[79]: returns           0.805002
         strat_log_reg     0.971623
         strat_gauss_nb    0.986420
         strat_svm         1.452406
         dtype: float64

In [80]: data[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 现在，使用 5 个滞后的对数收益率序列。

❷ 实值特征变换为二元数据。



图 15-13　欧元/美元与基于分类的交易策略（5 个二元滞后特征）的表现

15.6.3　5 个数字化特征

最后，以下代码使用历史对数收益率的一阶矩和二阶矩数字化特征数据，可以得到更多可能的特征值组合。这改善了所有分类算法的绩效，但 SVM 的改善仍然最为明显（见图 15-14）：

```
In [81]: mu = data['returns'].mean() ❶
         v = data['returns'].std() ❷

In [82]: bins = [mu - v, mu, mu + v] ❸
         bins ❸
Out[82]: [-0.006033537040418665, -0.00010174015279231306, 0.005830056734834039]

In [83]: create_bins(data, bins)

In [84]: data[cols_bin].head()
Out[84]:             lag_1_bin  lag_2_bin  lag_3_bin  lag_4_bin  lag_5_bin
         Date
         2010-01-12          3          3          0          2          1
         2010-01-13          1          3          3          0          2
         2010-01-14          2          1          3          3          0
         2010-01-15          1          2          1          3          3
         2010-01-19          0          1          2          1          3

In [85]: fit_models(data)

In [86]: derive_positions(data)
```

```
In [87]: evaluate_strats(data)

In [88]: data[sel].sum().apply(np.exp)
Out[88]: returns           0.805002
         strat_log_reg     1.431120
         strat_gauss_nb    1.815304
         strat_svm         5.653433
         dtype: float64

In [89]: data[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 使用平均对数收益率。

❷ 标准差。

❸ 数字化特征数据。

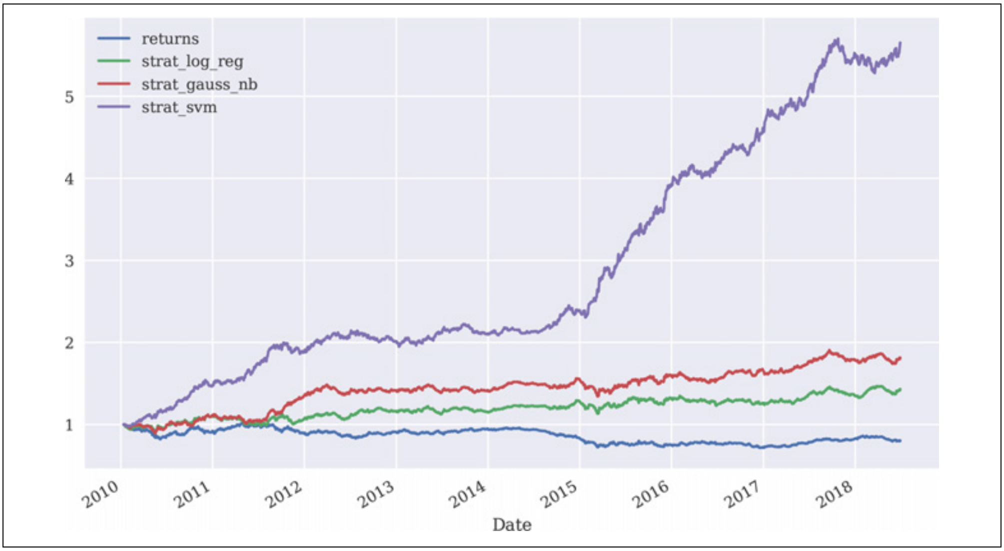


图 15-14　欧元/美元与基于分类的交易策略（5 个数字化滞后特征）的表现



特征类型

本章只使用滞后收益率数据作为特征数据，主要形式是二元特征或者数字化特征。这主要是为了方便，因为此类特征数据可以从金融时间序列获得。但是，在实际应用中，特征数据可以来自多种多样的数据源，可能包含其他金融时间序列和由此得出的统计量、宏观经济数据、公司金融指标或者新闻报道。对这一主题的深入讨论参见 López de Prado 的著作（2018）。也有一些用于自动化时间序列特征提取的 Python 库，如 `tsfresh`。

15.6.4　顺序训练-测试分离

为了更好地评判分类算法的绩效，接下来用下面的代码实现顺序训练-测试分离。这里的思路是模拟只有某一时间点前的数据可用于训练 ML 算法的情况。在实时交易期间，算法面对的是前所未见的数据，那正是算法证明其价值的场合。在这种特例中，所有分类算法的绩效都优于（维持之前的简化假设）被动的基准投资，但只有 GNB 和 LR 算法实现了绝对正收益（见图 15-15）：

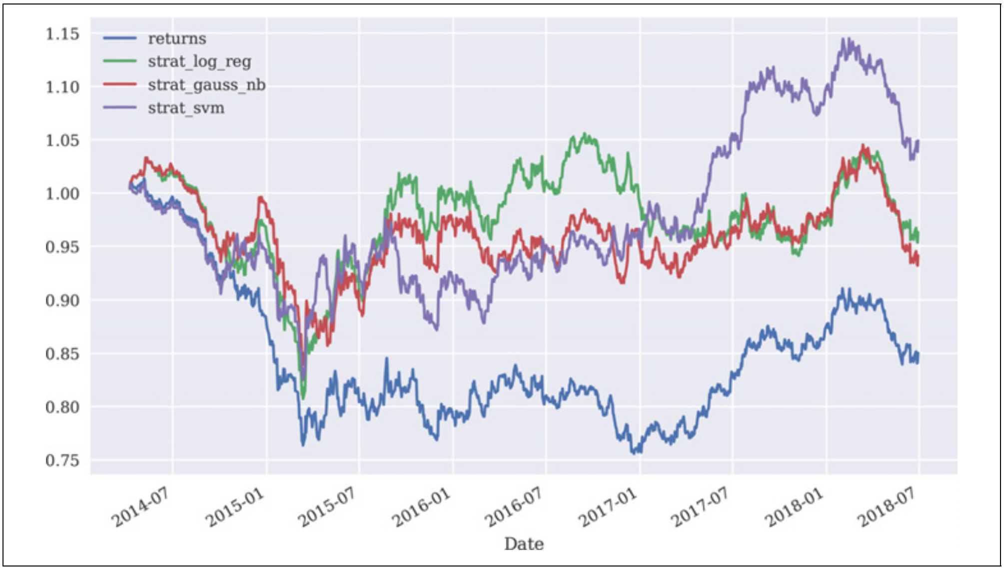


图 15-15　欧元/美元与基于分类的交易策略（顺序训练-测试分离）的表现

```
In [90]: split = int(len(data) * 0.5)

In [91]: train = data.iloc[:split].copy() ❶

In [92]: fit_models(train) ❶

In [93]: test = data.iloc[split:].copy() ❷

In [94]: derive_positions(test) ❷

In [95]: evaluate_strats(test) ❷

In [96]: test[sel].sum().apply(np.exp)
Out[96]: returns           0.850291
         strat_log_reg     0.962989
         strat_gauss_nb    0.941172
         strat_svm         1.048966
```

```
         dtype: float64

In [97]: test[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 在训练数据上训练所有分类算法。

❷ 在测试数据上测试所有分类算法。

15.6.5 随机训练-测试分离

在二元或者数字化特征数据上训练和测试分类算法，思路是使用特征值模式预测未来市场变动，可以有高于 50%的准确率。也就是说，这种方法假定模式的预测能力在一段时间内保持不变。从这个意义上说，不管使用哪一部分数据训练算法、哪一部分测试算法，都不应该造成（太大）差异，这意味着我们可以打破训练和测试数据的时间顺序。

典型的方法之一就是随机训练-测试分离，以样本外数据测试分类算法表现，同时试图模拟现实状况，交易期间算法将面对连续流入的新数据。我们采用的方法与 13.4.2 节中的相同。根据这种方法，SVM 算法再次得到最佳的样本外表现（见图 15-16）：

```
In [98]: from sklearn.model_selection import train_test_split

In [99]: train, test = train_test_split(data, test_size=0.5,
                                        shuffle=True, random_state=100)

In [100]: train = train.copy().sort_index() ❶

In [101]: train[cols_bin].head()
Out[101]:             lag_1_bin  lag_2_bin  lag_3_bin  lag_4_bin  lag_5_bin
          Date
          2010-01-12          3          3          0          2          1
          2010-01-13          1          3          3          0          2
          2010-01-14          2          1          3          3          0
          2010-01-15          1          2          1          3          3
          2010-01-20          1          0          1          2          1

In [102]: test = test.copy().sort_index() ❶

In [103]: fit_models(train)

In [104]: derive_positions(test)

In [105]: evaluate_strats(test)

In [106]: test[sel].sum().apply(np.exp)
Out[106]: returns          0.878078
          strat_log_reg    0.735893
```

```
               strat_gauss_nb     0.765009
               strat_svm          0.695428
               dtype: float64

    In [107]: test[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 复制训练和测试数据集并恢复时间顺序。



图 15-16　欧元/美元与基于分类的交易策略（随机训练-测试分离）的表现

15.7　深度神经网络

深度神经网络（DNN）试图模拟人类大脑的运作。它们通常由输入层（特征）、输出层（标签）和多个隐藏层组成。隐藏层的存在使神经网络变得更“深”，能学习更复杂的关系、在许多类问题上表现更好。应用 DNN 时，人们通常用“深度学习”代替“机器学习”。这一领域的介绍可参见 Geron（2017）或者 Gibson 和 Patterson（2017）的著作。

15.7.1　用 scikit-learn 实现 DNN

本节应用 scikit-learn 中的 MLPClassifier 算法（13.4.2 节中曾介绍过）。首先，它使用数字化特征在整个数据集上训练和测试。这个算法实现了极其出色的样本内表现（见图 15-17），说明了 DNN 在此类问题上的能力。但是，它也表现出了明显的过度拟合现象，因为表现之好确实有些不真实：

```
    In [108]: from sklearn.neural_network import MLPClassifier

    In [109]: model = MLPClassifier(solver='lbfgs', alpha=1e-5,
```

```
                                    hidden_layer_sizes=2 * [250],
                                    random_state=1)

In [110]: %time model.fit(data[cols_bin], data['direction'])
          CPU times: user 16.1 s, sys: 156 ms, total: 16.2 s
          Wall time: 9.85 s

Out[110]: MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
           beta_1=0.9,
                 beta_2=0.999, early_stopping=False, epsilon=1e-08,
                 hidden_layer_sizes=[250, 250], learning_rate='constant',
                 learning_rate_init=0.001, max_iter=200, momentum=0.9,
                 n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
                 random_state=1, shuffle=True, solver='lbfgs', tol=0.0001,
                 validation_fraction=0.1, verbose=False, warm_start=False)

In [111]: data['pos_dnn_sk'] = model.predict(data[cols_bin])

In [112]: data['strat_dnn_sk'] = data['pos_dnn_sk'] * data['returns']

In [113]: data[['returns', 'strat_dnn_sk']].sum().apply(np.exp)
Out[113]: returns          0.805002
          strat_dnn_sk     35.156677
          dtype: float64

In [114]: data[['returns', 'strat_dnn_sk']].cumsum().apply(
                    np.exp).plot(figsize=(10, 6));
```

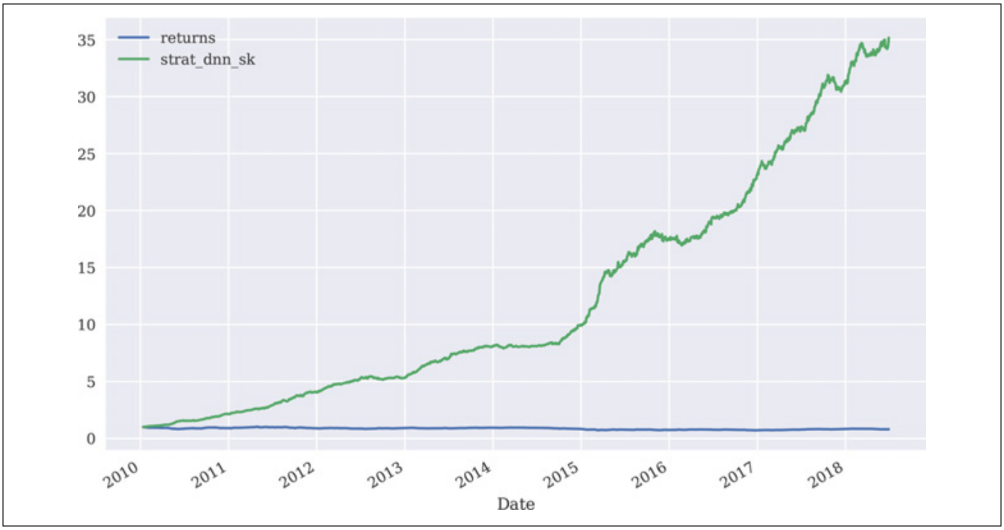


图 15-17　欧元/美元与基于 DNN 的交易策略（scikit-learn，样本内）的表现

为了避免 DNN 模型的过度拟合，接下来我们应用随机训练-测试分离。算法再一次胜过了被动基准投资，得到了绝对正效益（见图 15-18）。但是，结果现在似乎较为现实：



图 15-18　欧元/美元与基于 DNN 的交易策略（scikit-learn，随机训练-测试分离）的表现

```
In [115]: train, test = train_test_split(data, test_size=0.5,
                                         random_state=100)

In [116]: train = train.copy().sort_index()

In [117]: test = test.copy().sort_index()

In [118]: model = MLPClassifier(solver='lbfgs', alpha=1e-5, max_iter=500,
                                hidden_layer_sizes=3 * [500], random_
state=1) ❶

In [119]: %time model.fit(train[cols_bin], train['direction'])
          CPU times: user 2min 26s, sys: 1.02 s, total: 2min 27s
          Wall time: 1min 31s

Out[119]: MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
           beta_1=0.9,
               beta_2=0.999, early_stopping=False, epsilon=1e-08,
               hidden_layer_sizes=[500, 500, 500], learning_rate='constant',
               learning_rate_init=0.001, max_iter=500, momentum=0.9,
               n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
               random_state=1, shuffle=True, solver='lbfgs', tol=0.0001,
               validation_fraction=0.1, verbose=False, warm_start=False)
```

```
In [120]: test['pos_dnn_sk'] = model.predict(test[cols_bin])

In [121]: test['strat_dnn_sk'] = test['pos_dnn_sk'] * test['returns']

In [122]: test[['returns', 'strat_dnn_sk']].sum().apply(np.exp)
Out[122]: returns        0.878078
          strat_dnn_sk   1.242042
          dtype: float64

In [123]: test[['returns', 'strat_dnn_sk']].cumsum(
                      ).apply(np.exp).plot(figsize=(10, 6));
```

❶ 增大隐藏层和隐藏单元数量。

15.7.2 用 TensorFlow 实现 DNN

TensorFlow 已经成为流行的深度学习程序库，它由 Google 公司开发并提供支持，应用于多种多样的机器学习问题。Zedah 和 Ramsundar（2018）的著作深入介绍了深度学习中的 TensorFlow。与 scikit-learn 相同，利用 13.4.2 节的背景知识，应用 TensorFlow 的 DNNClassifier 算法得出算法交易策略十分简单。在样本内，该算法的效果好于被动基准投资，表现出可观的绝对收益（见图 15-19），这同样说明存在过度拟合：

```
In [124]: import tensorflow as tf
          tf.logging.set_verbosity(tf.logging.ERROR)

In [125]: fc = [tf.contrib.layers.real_valued_column('lags', dimension=lags)]

In [126]: model = tf.contrib.learn.DNNClassifier(hidden_units=3 * [500],
                                                  n_classes=len(bins) + 1,
                                                  feature_columns=fc)

In [127]: def input_fn():
              fc = {'lags': tf.constant(data[cols_bin].values)}
              la = tf.constant(data['direction'].apply(
                           lambda x: 0 if x < 0 else 1).values,
                           shape=[data['direction'].size, 1])
              return fc, la

In [128]: %time model.fit(input_fn=input_fn, steps=250) ❶
          CPU times: user 2min 7s, sys: 8.85 s, total: 2min 16s
          Wall time: 49 s

Out[128]: DNNClassifier(params={'head':
          <tensorflow.contrib.learn.python.learn.estimators.head._MultiClassHead
          object at 0x1a19acf898>, 'hidden_units': [500, 500, 500],
          'feature_columns': (_RealValuedColumn(column_name='lags', dimension=5,
```

```
                default_value=None, dtype=tf.float32, normalizer=None),), 'optimizer':
                None, 'activation_fn': <function relu at 0x1161441e0>, 'dropout':
                None, 'gradient_clip_norm': None, 'embedding_lr_multipliers': None,
                'input_layer_min_slice_size': None})

In [129]: model.evaluate(input_fn=input_fn, steps=1) ❷
Out[129]: {'loss': 0.6879357, 'accuracy': 0.5379925, 'global_step': 250}

In [130]: pred = np.array(list(model.predict(input_fn=input_fn))) ❷
          pred[:10] ❷
Out[130]: array([0, 0, 0, 0, 0, 1, 0, 1, 1, 0])

In [131]: data['pos_dnn_tf'] = np.where(pred > 0, 1, -1) ❸

In [132]: data['strat_dnn_tf'] = data['pos_dnn_tf'] * data['returns']

In [133]: data[['returns', 'strat_dnn_tf']].sum().apply(np.exp)
Out[133]: returns         0.805002
          strat_dnn_tf    2.437222
          dtype: float64

In [134]: data[['returns', 'strat_dnn_tf']].cumsum(
                      ).apply(np.exp).plot(figsize=(10, 6));
```

❶ 训练所需时间可能很可观。

❷ 二元预测（0,1）。

❸ 必须转换成市场头寸（−1，+1）。



图 15-19　欧元/美元与基于 DNN 的交易策略（TensorFlow，样本内）的表现

下面的代码再次实施随机训练-测试分离，得到基于 DDN 算法交易策略的更为真实的表现情况。正如预期，样本外的表现较差（见图 15-20）。此外，在特定参数下，TensorFlow 的 DNNClassifier 的表现比 scikit-learn MLPClassifier 算法的低几个百分点：

```
In [135]: model = tf.contrib.learn.DNNClassifier(hidden_units=3 * [500],
                                                   n_classes=len(bins) + 1,
                                                   feature_columns=fc)

In [136]: data = train

In [137]: %time model.fit(input_fn=input_fn, steps=2500)
          CPU times: user 11min 7s, sys: 1min 7s, total: 12min 15s
          Wall time: 4min 27s

Out[137]: DNNClassifier(params={'head':
          <tensorflow.contrib.learn.python.learn.estimators.head._MultiClassHead
          object at 0x116828cc0>, 'hidden_units': [500, 500, 500],
          'feature_columns': (_RealValuedColumn(column_name='lags', dimension=5,
          default_value=None, dtype=tf.float32, normalizer=None),), 'optimizer':
          None, 'activation_fn': <function relu at 0x1161441e0>, 'dropout':
          None, 'gradient_clip_norm': None, 'embedding_lr_multipliers': None,
          'input_layer_min_slice_size': None})

In [138]: data = test
In [139]: model.evaluate(input_fn=input_fn, steps=1)
Out[139]: {'loss': 0.82882184, 'accuracy': 0.48968107, 'global_step': 2500}

In [140]: pred = np.array(list(model.predict(input_fn=input_fn)))

In [141]: test['pos_dnn_tf'] = np.where(pred > 0, 1, -1)

In [142]: test['strat_dnn_tf'] = test['pos_dnn_tf'] * test['returns']

In [143]: test[['returns', 'strat_dnn_sk', 'strat_dnn_tf']].sum().apply(np.exp)
Out[143]: returns         0.878078
          strat_dnn_sk    1.242042
          strat_dnn_tf    1.063968
          dtype: float64

In [144]: test[['returns', 'strat_dnn_sk', 'strat_dnn_tf']].cumsum(
                      ).apply(np.exp).plot(figsize=(10, 6));
```

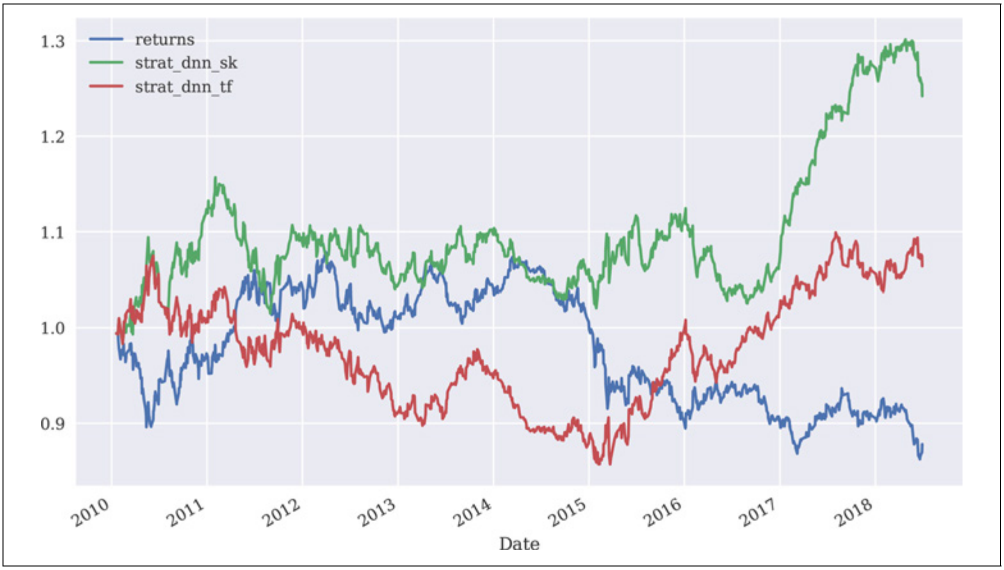


图 15-20　欧元/美元与基于 DNN 交易策略（TensorFlow，随机训练-测试分离）的表现

表现结果

迄今为止，向量化事后检验展现的不同算法交易策略的表现都仅用于说明目的。除了没有交易成本的简化假设之外，结果还取决于许多其他（大部分是随意选择的）参数。还有一个决定因素是我们自始至终都使用相对小的欧元/美元汇率日终价格数据集。本节的重点是阐述不同方法和 ML 算法在金融数据上的应用，而不是得出用于实际部署的可靠算法交易策略。第 16 章将讨论此类问题。

15.8　结语

本章介绍了算法交易策略，并根据向量化事后检验评判其表现。我们首先从基于两条简单移动平均线的简单算法交易策略入手，这是一种广为人知、在实践中应用数十年的策略。我们将这种算法用于阐述向量化事后检验，并大量使用了 NumPy 和 pandas 的向量化数据分析能力。本章还使用了 OLS 回归，以实时金融时间序列为基础介绍了随机游走假设。这是任何算法交易策略证明其价值时必备的基准。

本章的核心是 13.4 节中介绍的机器学习算法的应用。许多算法（大部分属于分类算法）的应用都基于基本相同的“节奏”。我们以滞后对数收益率数据的各种变形为特征——但这当然不是必然的约束，主要是为了方便和简单。此外，分析基于许多简化

假设，因为本章的重点主要是机器学习算法应用的技术特征，这些机器学习算法被应用到金融时间序列数据，以预测金融市场变动的方向。

15.9 延伸阅读

本章引用的论文如下：

- Brock, William, Josef Lakonishok, and Blake LeBaron (1992). “Simple Technical Trading Rules and the Stochastic Properties of Stock Returns.” *Journal of Finance*, Vol. 47, No. 5, pp. 1731-1764.
- Fama, Eugene (1965). “Random Walks in Stock Market Prices.” Selected Papers, No. 16, Graduate School of Business, University of Chicago.
- Jensen, Michael (1978). “Some Anomalous Evidence Regarding Market Efficiency.” *Journal of Financial Economics*, Vol. 6, No. 2/3, pp. 95-101.

介绍本章相关主题的金融学图书包括：

- Baxter, Martin, and Andrew Rennie (1996). *Financial Calculus*. Cambridge, England: Cambridge University Press.
- Chan, Ernest (2009). *Quantitative Trading*. Hoboken, NJ: John Wiley & Sons.
- Chan, Ernest (2013). *Algorithmic Trading*. Hoboken, NJ: John Wiley & Sons.
- Chan, Ernest (2017). *Machine Trading*. Hoboken, NJ: John Wiley & Sons.
- López de Prado, Marcos (2018). *Advances in Financial Machine Learning*. Hoboken, NJ: John Wiley & Sons.

介绍本章相关主题的技术图书包括：

- Albon, Chris (2018). *Machine Learning with Python Cookbook*. Sebastopol, CA: O’Reilly.
- Géron, Aurélien (2017). Hands-On Machine Learning with Scikit-Learn and Tensorflow. Sebastopol, CA: O’Reilly.
- Gibson, Adam, and Josh Patterson (2017). *Deep Learning*. Sebastopol, CA: O’Reilly.
- VanderPlas, Jake (2016). *Python Data Science Handbook*. Sebastopol, CA: O’Reilly.
- Zadeh, Reza Bosagh, and Bharath Ramsundar (2018). *TensorFlow for Deep Learning*. Sebastopol, CA: O’Reilly.

介绍 Python 算法交易的全面在线训练课程参见 Python Quants 官网。

第 16 章

自动化交易

人们担心计算机将变得太过聪明并接管这个世界，但真正的问题是它们太蠢，而且已经接管了世界。

——彼得·多明戈斯

人们可能会想："现在该怎么办？"交易平台已经存在，人们可以读取历史数据和流数据，下买卖单和检查账户状态。我们也已经介绍了通过预测市场价格变动方向来得出算法交易策略的不同方法。能不能把这些技术组合起来，以自动化的方式工作？这个问题不能笼统地回答。但是，本章将介绍一些在此背景下十分重要的主题。我们假定只部署一个自动化算法交易策略，这将简化资本和风险管理等方面的问题。

本章介绍如下内容。

资本管理

正如本节所述，依赖于策略特性及可用交易资本，凯利标准有助于控制交易规模。

基于 ML 的交易策略

为了获得对算法交易策略的信心，必须对其效果和风险特性进行全面的事后检验。我们使用的策略示例基于第 15 章中介绍的一种机器学习分类算法。

在线算法

为了部署自动化交易的算法交易策略，必须将其翻译成处理实时流入数据的在线算法。

基础设施与部署

为了稳定、可靠地运行自动化算法交易策略，从可用性、性能和安全角度看，云端部署都是非常可取的选择。

日志与监控

日志在自动化交易策略部署期间的历史与某些事件的分析上起到了重要作用；通过套接字通信进行监控，可以实时（远程）观察事件。

16.1 资本管理

算法交易的核心问题之一是，在给定的总可用资本中，要分配多少资本给指定的算法交易策略。这个问题的答案取决于交易者试图通过算法交易实现的主要目标。大部分个人和金融机构都认为，长期财富最大化是好的候选目标之一。这也是 Edward Thorpe 在导出投资的“凯利标准”时所想的。Rotando 和 Thorp 的论文（1992）介绍过这个算法。

16.1.1 二项设定中的凯利标准

介绍凯利标准投资理论的常见方法是以掷硬币游戏为基础，或者更普遍的说法——二项设定（只有两种可能结果）。本节也遵循这一路径。假定一位赌徒与具有无限财富的银行或者赌场玩掷硬币游戏。再假设硬币为正面的概率为 p（$0.5 < p < 1$），背面的概率为 $q = 1-p < 0.5$。这位赌徒可以下注 b（大于 0 的任何数），如果押注正确则赢得相同的金额，如果错误则输掉全部。因此，在单次设定下，这一赌局 B（表示该游戏的随机变量）的预期价值为：

$$E[B] = p \cdot b - q \cdot b = (p - q) \cdot b > 0$$

风险中立且具有无限资金的赌徒将会下尽可能大的赌注，这样可以最大化预期回报。然而，金融市场上的交易通常不是一次性博弈，而是重复进行的。因此，假定 b_i 代表第 i 天下的赌注，c_0 代表原始资本。根据当天下注的结果，第一天结束时的资本 c_1 可能是 c_0+b_1 或者 c_0-b_1。照此重复 n 次的赌局预期价值为：

$$E[B^n] = c_0 + \sum_{i=1}^{n} (p-q) \cdot b_i$$

在经典经济理论中，风险中立、预期效用最大化的条件下，赌徒将会努力最大化上述表达式的值。很容易看出，和一次性投注的情况类似，投入所有可用资金（即 $b_i = c_i-1$）可以达到目的。然而，这也意味着一次损失所有可用资金将归零，导致破产（除非可以无限借贷）。因此，这种策略不能得到长期财富的最大化。

投入最大可用资本可能导致突然破产，完全不下注则能避免任何类型的损失，但却不能从有利的赌局中得益。这就是凯利标准的用武之地，它可以得出每轮赌局中可用资本的最优比例 f^*。假定 $n = h + t$，h 代表 n 轮赌局中硬币正面向上的次数，而 t 代表背面向上的次数。按照这些定义，n 轮后的可用资本为：

$$c_n = c_0 \cdot (1+f)^h \cdot (1-f)^t$$

在这种背景下，长期财富最大化可以归结为最大化每次下注的平均几何增长率，公式如下：

$$
\begin{aligned}
r^g &= \log\left(\frac{c_n}{c_0}\right)^{1/n} \\
&= \log\left[\frac{c_0 \cdot (1+f)^h \cdot (1-f)^t}{c_0}\right]^{1/n} \\
&= \log[(1+f)^h \cdot (1-f)^t]^{1/n} \\
&= \frac{h}{n}\log(1+f) + \frac{t}{n}\log(1-f)
\end{aligned}
$$

此时，问题的形式变成通过选择最优的 f，来最大化预期平均增长率。由 $E[h] = n \cdot p$ 和 $E[t] = n \cdot q$，可以得出：

$$
\begin{aligned}
E[r^g] &= E\left[\frac{h}{n}\log(1+f) + \frac{t}{n}\log(1-f)\right] \\
&= E[p\log(1+f) + q\log(1-f)] \\
&= p\log(1+f) + q\log(1-f) \\
&\equiv G(f)
\end{aligned}
$$

现在，可以根据一阶条件选择最优比例 f^*来最大化这一项。一阶导数由以下公式算出：

$$
\begin{aligned}
G'(f) &= \frac{p}{1+f} - \frac{q}{1-f} \\
&= \frac{p - pf - q - qf}{(1+f)(1-f)} \\
&= \frac{p - q - f}{(1+f)(1-f)}
\end{aligned}
$$

从一阶条件可以得出：

$$
G'(f) \overset{!}{=} 0 \Rightarrow f^* = p - q
$$

如果相信这就是最大值（而不是最小值），这一结果意味着，每轮最优的投入比例是 $f^* = p - q$。例如，如果 $p = 0.55$，则 $f^* = 0.55-0.45 = 0.1$，表示最优投资比例为 10%。下面的 Python 代码通过模拟将这些概念与结果形式化：

首先是导入和配置：

```
In [1]: import math
        import time
        import numpy as np
        import pandas as pd
        import datetime as dt
        import cufflinks as cf
```

```
        from pylab import plt
In [2]: np.random.seed(1000)
        plt.style.use('seaborn')
        %matplotlib inline
```

我们的思路是模拟连续多次掷硬币（例如 50 个序列，每个序列 100 次）。Python 代码很简单：

```
In [3]: p = 0.55 ❶

In [4]: f = p - (1 - p) ❷

In [5]: f ❷
Out[5]: 0.10000000000000009

In [6]: I = 50 ❸

In [7]: n = 100 ❹
```

❶ 设置正面向上的固定概率。

❷ 根据凯利标准计算最优比例。

❸ 模拟的序列数量。

❹ 每次序列的试验次数。

重要的部分是 Python 函数 `run_simulation()`，它根据之前的假设实现模拟。图 16-1 展示了模拟结果：

```
In [8]: def run_simulation(f):
            c = np.zeros((n, I)) ❶
            c[0] = 100 ❷
            for i in range(I): ❸
                for t in range(1, n): ❹
                    o = np.random.binomial(1, p) ❺
                    if o > 0: ❻
                        c[t, i] = (1 + f) * c[t - 1, i] ❼
                    else: ❽
                        c[t, i] = (1 - f) * c[t - 1, i] ❾
            return c

In [9]: c_1 = run_simulation(f) ❿

In [10]: c_1.round(2)
Out[10]: array([[100. , 100. , 100. , ..., 100. , 100. , 100. ],
                [ 90. , 110. , 90. , ...,  110. ,  90. , 110. ],
                [ 99. , 121. , 99. , ...,  121. ,  81. , 121. ],
                ...,
```

```
                [226.35, 338.13, 413.27, ..., 123.97, 123.97, 123.97],
                [248.99, 371.94, 454.6 , ..., 136.37, 136.37, 136.37],
                [273.89, 409.14, 409.14, ..., 122.73, 150.01, 122.73]])

In [11]: plt.figure(figsize=(10, 6))
         plt.plot(c_1, 'b', lw=0.5) ⓫
         plt.plot(c_1.mean(axis=1), 'r', lw=2.5); ⓬
```

❶ 实例化一个 `ndarray` 对象，以保存模拟结果。

❷ 将起始资本设置为 100。

❸ 序列模拟的外循环。

❹ 序列模拟的内循环。

❺ 模拟掷硬币。

❻ 如果为 1（即正面向上）。

❼ 将赢取的赌金加入资本中。

❽ 如果为 0（即背面向上）。

❾ 从资本中减去损失。

❿ 运行模拟。

⓫ 绘制 50 个序列的图表。

⓬ 绘制全部 50 个序列的平均值。

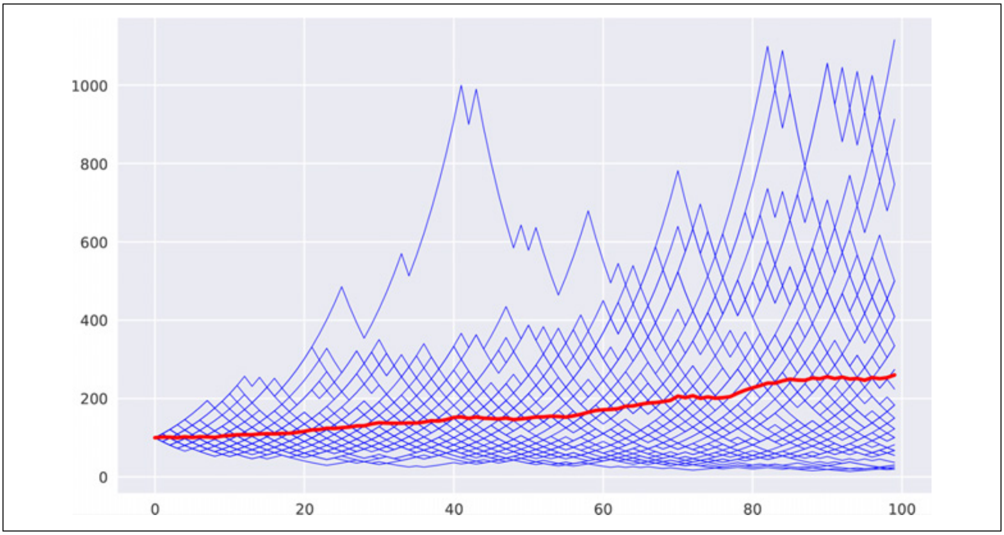


图 16-1　50 个模拟序列，每个序列 100 次试验（红线为平均线）

下面的代码用于对不同的 f 值重复模拟。如图 16-2 所示，较低的比例导致较低的平均增长率。较高的 f 值可能导致模拟结束时较高的平均资本（$f = 0.25$）或者低得多的平均资本（$f = 0.5$）。在这两种情况下，f 较高时波动率明显增加。

```
In [12]: c_2 = run_simulation(0.05) ❶

In [13]: c_3 = run_simulation(0.25) ❷

In [14]: c_4 = run_simulation(0.5) ❸

In [15]: plt.figure(figsize=(10, 6))
         plt.plot(c_1.mean(axis=1), 'r', label='$f^*=0.1$')
         plt.plot(c_2.mean(axis=1), 'b', label='$f=0.05$')
         plt.plot(c_3.mean(axis=1), 'y', label='$f=0.25$')
         plt.plot(c_4.mean(axis=1), 'm', label='$f=0.5$')
         plt.legend(loc=0);
```

❶ 模拟 $f = 0.05$ 的情况。

❷ 模拟 $f = 0.25$ 的情况。

❸ 模拟 $f = 0.5$ 的情况。

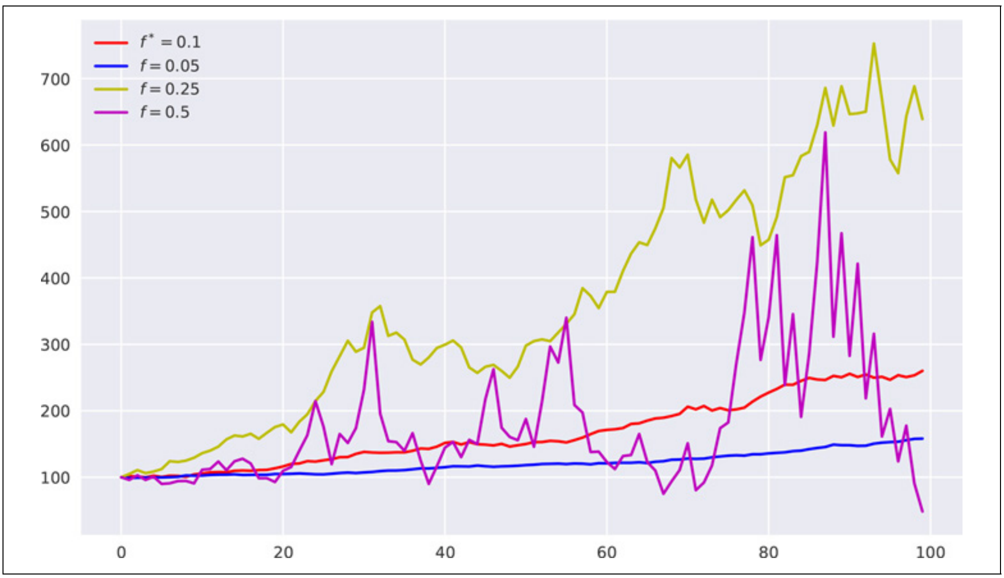


图 16-2　不同投资比例下一段时间的平均资本

16.1.2　用于股票及指数的凯利标准

假定现在有一个股票市场，在今日价值已知的情况下，相关股票（指数）在一年期限

之后只能取得两种价值。同样是二项设定，但这一次更接近股票市场真实情况的模型[1]。具体的假设如下：

$$P(r^S=\mu+\sigma)=P(r^S=\mu-\sigma)=\frac{1}{2}$$

设 $E[r^S]=\mu>0$ 是该股票一年内的预期收益率，$\sigma>0$ 是收益率标准差（波动率）。在单期设定中，我们可以得出一年以后的可用资本（c_0 与 f 的定义同前）：

$$c(f)=c_0\cdot[1+(1-f)\cdot r+f\cdot r^S]$$

式中，r 是资金不投资于股票时得到的固定短期利率。最大化几何增长率意味着最大化如下项：

$$G(f)=E\left[\log\frac{c(f)}{c_0}\right]$$

现在假定一年中有 n 个相关交易日，对于每个交易日：

$$P\left(r_i^S=\frac{\mu}{n}+\frac{\sigma}{\sqrt{n}}\right)=P\left(r_i^S=\frac{\mu}{n}-\frac{\sigma}{\sqrt{n}}\right)=\frac{1}{2}$$

注意，波动率与交易日数的平方根成正比。在这种假设下，每日价值加总即可得到全年收益：

$$c_n(f)=c_0\cdot\prod_{i=1}^{n}\left(1+(1-f)\cdot\frac{r}{n}+f\cdot r_i^S\right)$$

现在，我们必须最大化如下数量，才能在投资股票时实现长期财富最大化：

$$\begin{aligned}G_n(f)&=E\left[\log\frac{c_n(f)}{c_0}\right]\\&=E\left[\sum_{i=1}^{n}\log\left(1+(1-f)\cdot\frac{r}{n}+f\cdot r_i^S\right)\right]\\&=\frac{1}{2}\sum_{i=1}^{n}\log\left[1+(1-f)\cdot\frac{r}{n}+f\cdot\left(\frac{\mu}{n}+\frac{\sigma}{\sqrt{n}}\right)\right]\\&\quad+\log\left[1+(1-f)\cdot\frac{r}{n}+f\cdot\left(\frac{\mu}{n}-\frac{\sigma}{\sqrt{n}}\right)\right]\\&=\frac{n}{2}\log\left[\left(1+(1-f)\cdot\frac{r}{n}+f\cdot\frac{\mu}{n}\right)^2-\frac{f^2\sigma^2}{n}\right]\end{aligned}$$

1 这一论述遵循 Hung 的著作（2010）。——原注

使用泰勒级数展开，最终可以得到：

$$G_n(f) = r + (\mu - r) \cdot f - \frac{\sigma^2}{2} \cdot f^2 + \sigma\left(\frac{1}{\sqrt{n}}\right)$$

或者将其用于无限个交易时间点——即连续交易：

$$G_\infty(f) = r + (\mu - r) \cdot f - \frac{\sigma^2}{2} \cdot f^2$$

最优投资比例 f^*通过一阶条件给出，如以下的表达式：

$$f^* = \frac{\mu - r}{\sigma^2}$$

也就是用股票对无风险利率的超额收益除以收益率方差。这一表达式看起来与夏普指数类似（见 13.1 节），但不一样。

我们将用一个实例来说明这些公司的应用，以及它们在为交易策略配置权益类资产时的作用。例子中考虑的交易策略是简单地被动持有标普 500 指数看多头寸。为此，我们可以很快地读取基础数据，并很容易得出所需的统计量：

```
In [16]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                           index_col=0, parse_dates=True)

In [17]: symbol = '.SPX'

In [18]: data = pd.DataFrame(raw[symbol])

In [19]: data['returns'] = np.log(data / data.shift(1))

In [20]: data.dropna(inplace=True)

In [21]: data.tail()
Out[21]:                .SPX   returns
         Date
         2018-06-25  2717.07 -0.013820
         2018-06-26  2723.06  0.002202
         2018-06-27  2699.63 -0.008642
         2018-06-28  2716.31  0.006160
         2018-06-29  2718.37  0.000758
```

标普 500 指数在一段时间里的统计属性表明，投资于该指数看多头寸的最优比例大约为 4.5。换言之，对于每 1 美元的可用资金，应该投入 4.5 美元——意味着根据最优的凯利“比例”（或者在这种情况下称作“因数”），杠杆比率为 4.5。其他条件

不变的情况下，凯利标准表明，杠杆比率越高，预期收益率越高、波动率（方差）越低：

```
In [22]: mu = data.returns.mean() * 252 ❶

In [23]: mu ❶
Out[23]: 0.09898579893004976

In [24]: sigma = data.returns.std() * 252 ** 0.5 ❷

In [25]: sigma ❷
Out[25]: 0.1488567510081967

In [26]: r = 0.0 ❸

In [27]: f = (mu - r) / sigma ** 2 ❹

In [28]: f ❹
Out[28]: 4.4672043679706865
```

❶ 计算年化收益率。

❷ 计算年化波动率。

❸ 将无风险利率设置为 0（简单起见）。

❹ 计算投资于该策略的最优凯利比例。

下面的代码用于模拟凯利标准及最优杆杆比率的应用。为了简单和对比，初始保证金设置为 1，初始投入总资本设置为 $1 \cdot f^*$。根据这一策略动用资金的绩效，总资本每日按照可用保证金调整。若发生损失，则资本减少；盈利则资本增加。股票头寸与指数本身发展的对比如图 16-3 所示：

```
In [29]: equs = []

In [30]: def kelly_strategy(f):
             global equs
             equ = 'equity_{:.2f}'.format(f)
             equs.append(equ)
             cap = 'capital_{:.2f}'.format(f)
             data[equ] = 1 ❶
             data[cap] = data[equ] * f ❷
         for i, t in enumerate(data.index[1:]):
             t_1 = data.index[i] ❸
             data.loc[t, cap] = data[cap].loc[t_1] * \
                                math.exp(data['returns'].loc[t]) ❹
```

```
            data.loc[t, equ] = data[cap].loc[t] - \
                               data[cap].loc[t_1] + \
                               data[equ].loc[t_1] ❺
            data.loc[t, cap] = data[equ].loc[t] * f ❻

In [31]: kelly_strategy(f * 0.5) ❼

In [32]: kelly_strategy(f * 0.66) ❽

In [33]: kelly_strategy(f) ❾

In [34]: print(data[equs].tail())
                    equity_2.23  equity_2.95  equity_4.47
         Date
         2018-06-25     4.707070     6.367340     8.794342
         2018-06-26     4.730248     6.408727     8.880952
         2018-06-27     4.639340     6.246147     8.539593
         2018-06-28     4.703365     6.359932     8.775296
         2018-06-29     4.711332     6.374152     8.805026

In [35]: ax = data['returns'].cumsum().apply(np.exp).plot(legend=True,
                                                     figsize=(10, 6))
         data[equs].plot(ax=ax, legend=True);
```

❶ 为保证金（equity）生成新列，将初始值设置为 1。

❷ 为资本（capital）生成一个新列，将初始值设置为 $1 \cdot f^*$。

❸ 为前值选择合适的 `DatetimeIndex` 值。

❹ 根据收益率计算新的资本头寸。

❺ 根据资本头寸绩效调整保证金值。

❻ 根据新的保证金头寸和固定杠杆比例来调整资本头寸。

❼ 模拟基于凯利标准策略，将 f 减半。

❽ 使用 f 的 2/3。

❾ 使用 f。

如图 16-3 所示，应用最优凯利杠杆可能导致保证金头寸的不稳定变化（高波动率），这在杠杆比率为 4.47 下可以直观地看出。可以预期，随着杠杆的增加，保证金头寸的波动性也增大。因此，从业者往往降低杠杆，例如“凯利杠杆的一半”——在当前的例子中为 $0.5 \cdot f^* \approx 2.23$。图 16-3 还表明，保证金头寸值的变化低于“凯利杠杆”。随着 f 值的降低，风险确实减小了。

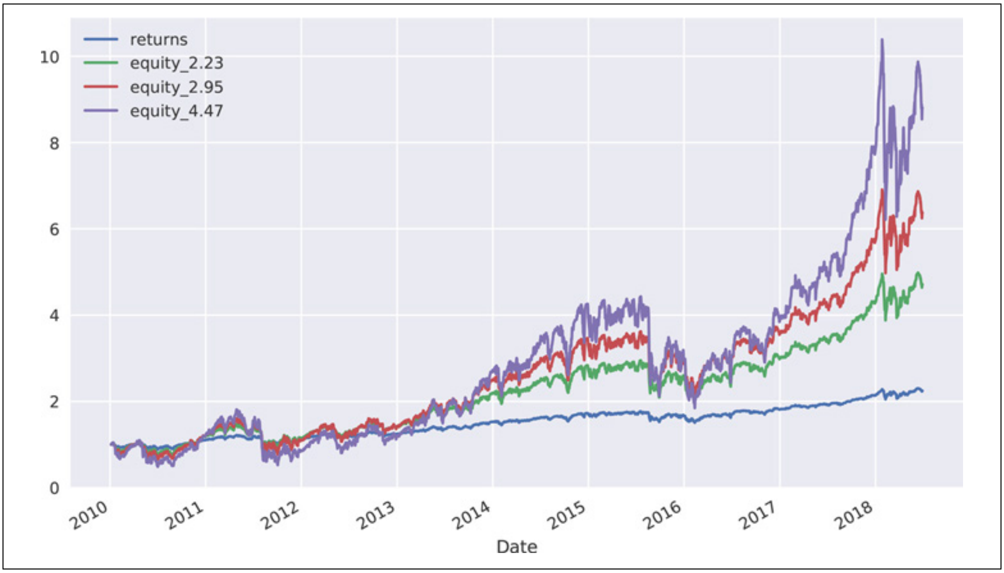


图 16-3　在不同 *f* 值下，标普 500 累计绩效与保证金头寸的对比

16.2　基于 ML 的交易策略

第 14 章介绍了 FXCM 交易平台、其 REST API 和 Python 包装器库 fxcmpy。本节将预测市场价格变动方向的基于 ML 的方法与 FXCM REST API 提供的历史数据结合，对欧元/美元货币组合的算法交易策略进行事后检验。我们使用向量化事后检验，这一次还考虑了比例交易成本——买卖价差。与第 15 章中介绍的朴素向量化事后检验方法相比，我们还加入了对所检验交易策略风险特性的更深入分析。

16.2.1　向量化事后检验

事后检验基于日内数据，确切地说基于 5 分钟线数据。下面的代码可以连接到 FXCM REST API，读取整个月的 5 分钟分时数据。图 16-4 显示了检索数据期间的中间收盘价。

```
In [36]: import fxcmpy

In [37]: fxcmpy.__version__
Out[37]: '1.1.33'

In [38]: api = fxcmpy.fxcmpy(config_file='../fxcm.cfg') ❶

In [39]: data = api.get_candles('EUR/USD', period='m5',
                                start='2018-06-01 00:00:00',
```

```
                                    stop='2018-06-30 00:00:00') ❶

In [40]: data.iloc[-5:, 4:]
Out[40]:                      askopen askclose askhigh asklow tickqty
         date
         2018-06-29 20:35:00 1.16862  1.16882 1.16896 1.16839    601
         2018-06-29 20:40:00 1.16882  1.16853 1.16898 1.16852    387
         2018-06-29 20:45:00 1.16853  1.16826 1.16862 1.16822    592
         2018-06-29 20:50:00 1.16826  1.16836 1.16846 1.16819    842
         2018-06-29 20:55:00 1.16836  1.16861 1.16876 1.16834    540

In [41]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 6083 entries, 2018-06-01 00:00:00 to 2018-06-
29 20:55:00
         Data columns (total 9 columns):
         bidopen     6083 non-null float64
         bidclose    6083 non-null float64
         bidhigh     6083 non-null float64
         bidlow      6083 non-null float64
         askopen     6083 non-null float64
         askclose    6083 non-null float64
         askhigh     6083 non-null float64
         asklow      6083 non-null float64
         tickqty     6083 non-null int64
         dtypes: float64(8), int64(1)
         memory usage: 475.2 KB

In [42]: spread = (data['askclose'] - data['bidclose']).mean() ❷
         spread ❷
Out[42]: 2.6338977478217845e-05

In [43]: data['midclose'] = (data['askclose'] + data['bidclose']) / 2 ❸

In [44]: ptc = spread / data['midclose'].mean() ❹
         ptc ❹
Out[44]: 2.255685318140426e-05

In [45]: data['midclose'].plot(figsize=(10, 6), legend=True);
```

❶ 连接到 API 并读取数据。

❷ 计算平均买卖价差。

❸ 由收盘买入与卖出价算出收盘中间价。

❹ 根据平均价差和平均收盘中间价，计算平均比例交易成本。

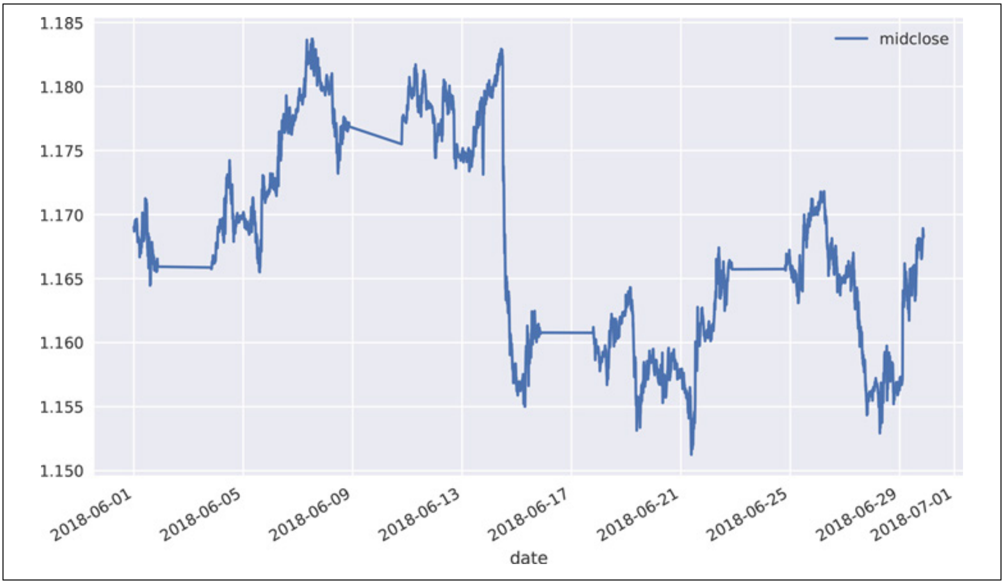


图 16-4　欧元/美元汇率（5 分钟线）

基于 ML 的策略以二元化滞后收益率数据为基础。换言之，ML 算法从向上及向下变动的历史模式中学习，确定更有可能发生哪一个方向的变动。相应地，以下代码创建值为 0 和 1 的特征数据，以及值为 1 和−1 的标签数据（表示观测到的市场变动方向）：

```
In [46]: data['returns'] = np.log(data['midclose'] / data['midclose'].shift(1))

In [47]: data.dropna(inplace=True)

In [48]: lags = 5

In [49]: cols = []
         for lag in range(1, lags + 1):
             col = 'lag_{}'.format(lag)
             data[col] = data['returns'].shift(lag) ❶
             cols.append(col)

In [50]: data.dropna(inplace=True)

In [51]: data[cols] = np.where(data[cols] > 0, 1, 0) ❷
```

```
In [52]: data['direction'] = np.where(data['returns'] > 0, 1, -1) ❸

In [53]: data[cols + ['direction']].head()
Out[53]:                      lag_1 lag_2 lag_3 lag_4 lag_5 direction
         date
         2018-06-01 00:30:00      1     0     1     0     1         1
         2018-06-01 00:35:00      1     1     0     1     0         1
         2018-06-01 00:40:00      1     1     1     0     1         1
         2018-06-01 00:45:00      1     1     1     1     0         1
         2018-06-01 00:50:00      1     1     1     1     1        -1
```

❶ 按照滞后期数，创建滞后收益率数据。

❷ 将特征值转换为二元数据。

❸ 将收益率数据转换为方向标签数据。

给定特征及标签数据之后，就可以应用不同的有监督学习算法了。在下文中，我们使用来自 scikit-learn 库的支持向量机算法。代码以顺序训练-测试分离为基础，来训练和测试算法交易策略。该模型在训练和测试数据上的准确率略高于 50%，测试数据上的准确率甚至还更高一些。在金融交易环境中，人们可能会用交易策略的"命中率"代替准确率，也就是盈利交易数量与全部交易数量的比值。由于命中率高于 50%，这可能表示，凯利标准背景与随机游走设定相比略占优势：

```
In [54]: from sklearn.svm import SVC
         from sklearn.metrics import accuracy_score

In [55]: model = SVC(C=1, kernel='linear', gamma='auto')

In [56]: split = int(len(data) * 0.80)

In [57]: train = data.iloc[:split].copy()

In [58]: model.fit(train[cols], train['direction'])
Out[58]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
         decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)

In [59]: accuracy_score(train['direction'], model.predict(train[cols])) ❶
Out[59]: 0.5198518823287389

In [60]: test = data.iloc[split:].copy()
```

```
In [61]: test['position'] = model.predict(test[cols])

In [62]: accuracy_score(test['direction'], test['position']) ❷
Out[62]: 0.5419407894736842
```

❶ 从样本内（训练数据）训练模型得出的预测准确率。

❷ 从样本外（测试数据）训练模型得出的预测准确率。

众所周知，命中率只是金融交易成功的一个方面。此外，交易策略隐含的交易成本和正确进行重要交易也是至关重要的[1]。为此，只有正规的向量化事后检验方法才能够评判交易策略的质量。下面的代码考虑基于平均买卖价差的比例交易成本。图 16-5 对比了算法交易策略（不考虑和考虑比例交易成本）与被动基准投资的效果：

```
In [63]: test['strategy'] = test['position'] * test['returns'] ❶

In [64]: sum(test['position'].diff() != 0) ❷
Out[64]: 660

In [65]: test['strategy_tc'] = np.where(test['position'].diff() != 0,
                                        test['strategy'] - ptc, ❸
                                        test['strategy'])

In [66]: test[['returns', 'strategy', 'strategy_tc']].sum(
                 ).apply(np.exp)
Out[66]: returns        0.999324
         strategy       1.026141
         strategy_tc    1.010977
         dtype: float64

In [67]: test[['returns', 'strategy', 'strategy_tc']].cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6));
```

❶ 得出基于 ML 算法交易策略的对数收益率。

❷ 根据头寸变化，计算交易策略产生的交易次数。

❸ 每当交易进行，从策略当天的对数收益率中扣除比例交易成本。

---

1 对于投资和交易绩效而言，正确把握最大的市场变动（即最大的向上及向下运动）至关重要，这是已经验证过的事实。图 16-5 和图 16-7 清晰地表示了这一特征。图中显示，该交易策略错过了标的金融工具的大幅上涨，导致交易策略出现绩效下滑。——原注



图 16-5　欧元/美元汇率与算法交易策略效果

向量化事后检验的局限性

向量化事后检验在测试市场现实与策略间接近程度方面有局限性。例如，它不能直接包含每次交易的固定成本。作为近似计算手段，我们可以取平均比例交易成本（基于平均仓位）的倍数，间接地考虑固定交易成本。但是，这通常不精确。如果需要更高的精确度，就需要其他方法，例如基于事件事后检验，这种方法明确地循环读取每一时刻的价格数据。

16.2.2　最优杠杆

有了交易策略的对数收益率数据后，我们就可以计算均值和方差，以便根据凯利标准得出最优杠杆。下面的代码将上述数值按比例调整为年化值，但是这并不会改变通过凯利标准计算出的杠杆值，因为收益率均值与方差以同样的比例调整：

```
In [68]: mean = test[['returns', 'strategy_tc']].mean() * len(data) * 12 ❶
         mean
Out[68]: returns       -0.040535
         strategy_tc    0.654711
         dtype: float64

In [69]: var = test[['returns', 'strategy_tc']].var() * len(data) * 12 ❷
```

```
         var
Out[69]: returns        0.007861
         strategy_tc    0.007837
         dtype: float64

In [70]: vol = var ** 0.5 ❸
         vol
Out[70]: returns        0.088663
         strategy_tc    0.088524
         dtype: float64

In [71]: mean / var ❹
Out[71]: returns       -5.156448
         strategy_tc   83.545792
         dtype: float64

In [72]: mean / var * 0.5 ❺
Out[72]: returns         -2.578224
         strategy_tc     41.772896
         dtype: float64
```

❶ 年化收益率均值。

❷ 年化方差。

❸ 年化波动率。

❹ 根据凯利标准（完整凯利）算出的最优杠杆。

❺ 根据凯利标准（半凯利）算出的最优杠杆。

使用“半凯利”标准，该交易策略的最优杠杆大约为 40。在许多经纪商（如 FXCM）和金融工具（如外汇及差价合约 CFD）上，即便是散户也可以实现这样的杠杆比率[1]。图 16-6 展示了该交易策略（含交易成本）在不同杠杆值下的绩效：

```
In [73]: to_plot = ['returns', 'strategy_tc']

In [74]: for lev in [10, 20, 30, 40, 50]:
             label = 'lstrategy_tc_%d' % lev
             test[label] = test['strategy_tc'] * lev ❶
             to_plot.append(label)

In [75]: test[to_plot].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

1 杠杆显著增大了与交易策略相关的风险。交易者应该小心阅读风险免责声明及规章。事后检验中的正效益也不能保证未来的绩效。这里看到的所有结果仅为了说明编程与分析方法的应用。在某些司法管辖区域（如德国），杠杆比率会根据不同组金融工具来对散户进行限制。——原注

❶ 对不同杠杆值下的策略收益进行比例调整。

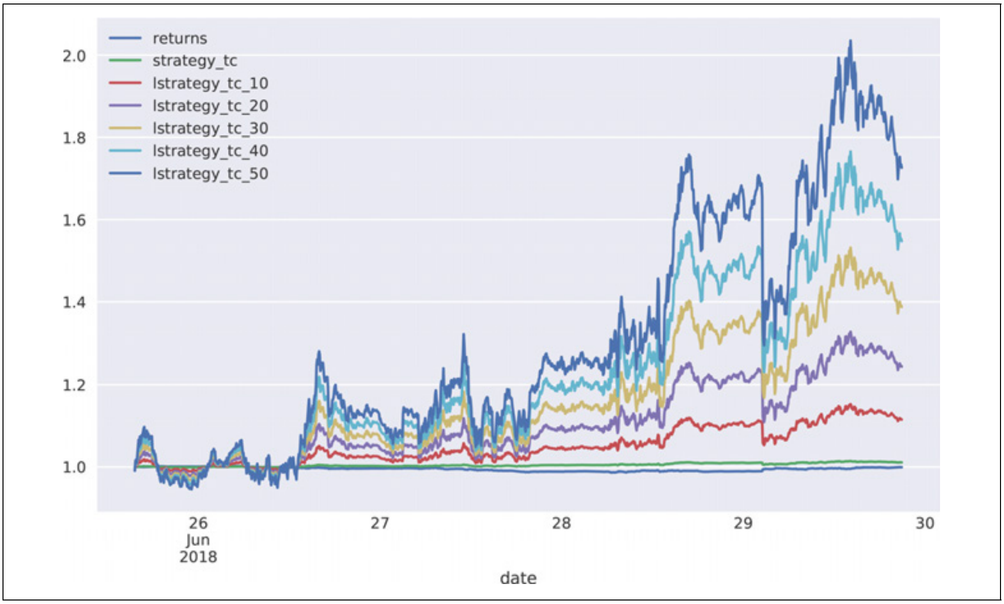


图 16-6　算法交易策略在不同杠杆值下的绩效

16.2.3　风险分析

由于杠杆显著地增加了与交易策略相关的风险，所以进行更深入的风险分析似乎是合适的举措。下面的风险分析假定杠杆比率为 30。首先，计算最大跌幅和最长下跌期间。最大跌幅是最近高点之后的最大损失（下跌）。相应地，最长下跌期间是交易策略回到最近高点所需的最长时间。分析假定初始保证金头寸为 3333 欧元，在杠杆比率为 30 的情况下，初始仓位约为 10 万欧元。我们还假设，不管绩效如何，一段时间内保证金不做任何调整：

```
In [76]: equity = 3333 ❶

In [77]: risk = pd.DataFrame(test['lstrategy_tc_30']) ❷

In [78]: risk['equity'] = risk['lstrategy_tc_30'].cumsum(
                 ).apply(np.exp) * equity ❸

In [79]: risk['cummax'] = risk['equity'].cummax() ❹

In [80]: risk['drawdown'] = risk['cummax'] - risk['equity'] ❺

In [81]: risk['drawdown'].max() ❻
```

```
Out[81]: 781.7073602069818

In [82]: t_max = risk['drawdown'].idxmax() ❼
         t_max ❼
Out[82]: Timestamp('2018-06-29 02:45:00')
```

❶ 初始保证金。

❷ 相关对数收益率时间序列。

❸ 根据初始保证金做比例调整。

❹ 一段时间内的累计最大价值。

❺ 一段时间内的下跌幅度。

❻ 最大跌幅值。

❼ 发生的时间点。

从技术上说，（新）高点的特征是跌幅值为 0。下跌期间是两个此类高点之间的时间。图 16-7 可视化了最大跌幅与下跌期间：

```
In [83]: temp = risk['drawdown'][risk['drawdown'] == 0] ❶

In [84]: periods = (temp.index[1:].to_pydatetime() -
                    temp.index[:-1].to_pydatetime()) ❷

In [85]: periods[20:30] ❷
Out[85]: array([datetime.timedelta(seconds=68700),
                datetime.timedelta(seconds=72000),
         datetime.timedelta(seconds=1800), datetime.timedelta(seconds=300),
         datetime.timedelta(seconds=600), datetime.timedelta(seconds=300),
                datetime.timedelta(seconds=17400),
         datetime.timedelta(seconds=4500), datetime.timedelta(seconds=1500),
                datetime.timedelta(seconds=900)], dtype=object)

In [86]: t_per = periods.max() ❸

In [87]: t_per ❸
Out[87]: datetime.timedelta(seconds=76500)

In [88]: t_per.seconds / 60 / 60 ❹
Out[88]: 21.25

In [89]: risk[['equity', 'cummax']].plot(figsize=(10, 6))
         plt.axvline(t_max, c='r', alpha=0.5);
```

❶ 确定高点，其跌幅值必须为 0。

❷ 计算所有高点之间的 `timedelta` 值。

❸ 以秒表示的最长下跌期。

❹ 以小时表示的最长下跌期。

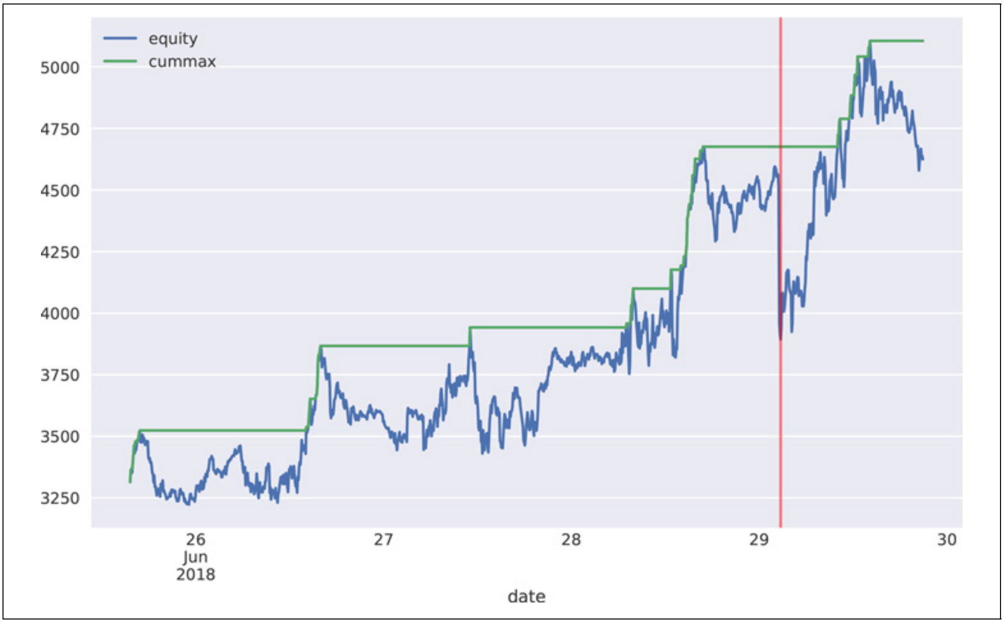


图 16-7　最大跌幅（垂直线）和下跌期间（水平线）

另一个重要风险指标是风险价值（VaR），它用金额表示，代表着某一时段和置信度下的预期最大损失。下面的代码根据一段时间内杠杆化交易策略保证金头寸的对数收益率，求得不同置信度下的 VaR。时间间隔固定为 5 分钟：

```
In [91]: import scipy.stats as scs

In [92]: percs = np.array([0.01, 0.1, 1., 2.5, 5.0, 10.0]) ❶

In [93]: risk['returns'] = np.log(risk['equity'] /
                                  risk['equity'].shift(1))

In [94]: VaR = scs.scoreatpercentile(equity * risk['returns'], percs) ❷

In [95]: def print_var():
             print('%16s %16s' % ('Confidence Level', 'Value-at-Risk'))
             print(33 * '-')
             for pair in zip(percs, VaR):
```

```
                print('%16.2f %16.3f' % (100 - pair[0], -pair[1])) ❸

In [96]: print_var() ❸
         Confidence Level    Value-at-Risk
         ---------------------------------
                    99.99          400.854
                    99.90          175.932
                    99.00           88.139
                    97.50           60.485
                    95.00           45.010
                    90.00           32.056
```

❶ 定义要使用的百分比值。

❷ 根据百分比值计算 VaR。

❸ 将百分比值转换为置信度，VaR 值（负值）转换为正值，以便打印。

最后，如下代码通过对原始 DataFrame 对象重新采样，计算一个小时的 VaR 值。实际上，除了最高置信度之外，其他置信度下的 VaR 值都增大了：

```
In [97]: hourly = risk.resample('1H', label='right').last() ❶

In [98]: hourly['returns'] = np.log(hourly['equity'] /
                                 hourly['equity'].shift(1))

In [99]: VaR = scs.scoreatpercentile(equity * hourly['returns'], percs) ❷

In [100]: print_var()
          Confidence Level   Value-at-Risk
          ---------------------------------
                    99.99          389.524
                    99.90          372.657
                    99.00          205.662
                    97.50          186.999
                    95.00          164.869
                    90.00          101.835
```

❶ 重新采样数据，将 5 分钟分时数据转换为 1 小时分时数据。

❷ 用重新采样的数据再次计算 VaR 值。

16.2.4 持久化模型对象

一旦根据事后检验、杠杆化和风险分析结果“接受”了算法交易策略，那么模型对象就可以持久化以供以后部署了。现在，它体现了基于 ML 的交易策略或者交易算法：

```
In [101]: import pickle

In [102]: pickle.dump(model, open('algorithm.pkl', 'wb'))
```

16.3　在线算法

目前为止测试的交易算法是离线算法。这种算法使用完整的数据集解决手头的问题。我们的问题是根据二元特征数据和方向标签数据训练一个 SVM 算法。实践中，在金融市场中部署交易算法时，它必须使用逐步送代的数据，以预测下一时间间隔的市场变动方向。本节使用 16.3 节的持久化模型对象，并将其嵌入流数据环境。将离线交易算法转换为在线交易算法的代码主要处理以下问题。

分笔交易数据

　　分笔交易数据实时送达，实时处理。

重新采样

　　分笔交易数据重新采样为适用于交易算法的分时区间。

预测

　　交易算法生成未来相关时间段市场变动方向的预测。

交易订单

　　按照当前头寸和算法生成的预测（信号），下单或者保持头寸。

14.3.2 节介绍了从 FXCM REST API 中实时读取分笔数据的方法。基本方法是订阅某个市场数据流，并传递一个处理数据的回调函数。

首先，加载持久化交易策略——它代表着交易逻辑。定义一个助手函数以在交易算法进行交易时打印开仓头寸也可能很有用：

```
In [103]: algorithm = pickle.load(open('algorithm.pkl', 'rb'))

In [104]: algorithm
Out[104]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape='ovr', degree=3, gamma='auto',
          kernel='linear', max_iter=-1, probability=False,
          random_state=None, shrinking=True, tol=0.001, verbose=False)

In [105]: sel = ['tradeId', 'amountK', 'currency',
                 'grossPL', 'isBuy'] ❶

In [106]: def print_positions(pos):
```

```
            print('\n\n' + 50 * '=')
            print('Going {}.\n'.format(pos))
            time.sleep(1.5) ❷
            print(api.get_open_positions()[sel]) ❸
            print(50 * '=' + '\n\n')
```

❶ 定义显示的 `DataFrame` 列。

❷ 等待订单执行并将其反映在开仓头寸中。

❸ 打印开仓头寸。

在定义和启动在线算法之前，设置几个参数：

```
In [107]: symbol = 'EUR/USD' ❶
          bar = '15s' ❷
          amount = 100 ❸
          position = 0 ❹
          min_bars = lags + 1 ❺
          df = pd.DataFrame() ❻
```

❶ 交易的金融工具代码。

❷ 重新采样分时长度；为了易于测试，分时长度可以比实际部署长度缩短一些（例如，用 15 秒而不是 15 分钟）。

❸ 交易数量（1000 的倍数）。

❹ 初始头寸（中立）。

❺ 实现第一次预测和交易所需的最小重采样时段数量。

❻ 以后用于重采样数据的空 `DataFrame` 对象。

下面是回调函数 `automated_strategy()`，它将交易算法转换为在实时环境中使用：

```
In [108]: def automated_strategy(data, dataframe):
              global min_bars, position, df
              ldf = len(dataframe) ❶
              df = dataframe.resample(bar, label='right').last().ffill() ❷
              if ldf % 20 == 0:
                 print('%3d' % len(dataframe), end=',')

              if len(df) > min_bars:
                  min_bars = len(df)
                  df['Mid'] = df[['Bid', 'Ask']].mean(axis=1)
                  df['Returns'] = np.log(df['Mid'] / df['Mid'].shift(1))
                  df['Direction'] = np.where(df['Returns'] > 0, 1, -1)
                  features = df['Direction'].iloc[-(lags + 1):-1] ❸
                  features = features.values.reshape(1, -1) ❹
```

```
            signal = algorithm.predict(features)[0] ❺

            if position in [0, -1] and signal == 1: ❻
                api.create_market_buy_order(
                    symbol, amount - position * amount)
                position = 1
                print_positions('LONG')

            elif position in [0, 1] and signal == -1: ❼
                api.create_market_sell_order(
                    symbol, amount + position * amount)
                position = -1
                print_positions('SHORT')

        if len(dataframe) > 350: ❽
            api.unsubscribe_market_data('EUR/USD')
            api.close_all()
```

❶ 获取包含分笔数据的 `DataFrame` 对象长度。

❷ 重新根据定义分时长度以重新采样分笔数据。

❸ 选择所有滞后期数的相关特征值。

❹ 将其重整为模型可用于预测的形式。

❺ 生成预测值（+1 或–1）。

❻ 进入（或保持）多头头寸的条件。

❼ 进入（或保持）空头头寸的条件。

❽ 停止交易、全部平仓的条件（根据读取的分笔交易数任意定义）。

16.4　基础设施与部署

部署实盘自动化算法交易策略需要合适的基础设施。这种基础设施应该满足如下条件。

可靠性

　　部署算法交易策略的基础设施应该具备高可用性（例如，>99.9%），还应该注意可靠性（自动备份、驱动器与 Web 连接冗余等）。

性能

　　根据处理的数据量和算法产生的计算需求，基础设施必须有足够的 CPU 核心、工作内存（RAM）和存储（SSD）；此外，Web 连接应该足够快。

安全性

操作系统及在其上运行的应用应该有强密码及 SSL 加密保护；硬件应该有防火、防水保护，并避免未授权物理访问。

本质上，这些需求只能通过从专业数据中心或云提供商租赁合适的基础设施来实现。通常只有金融市场上较大（甚至是最大）的参与者，投资满足上述需求的物理基础设施才是划算的。

从开发和测试的角度看，即便 DigitalOcean 最小的 Droplet（云实例）也足以完成入门工作了。在本书写作期间，这样的 Droplet 每月成本为 5 美元；使用费按小时计算，创建一台服务器只需要几分钟，将其销毁则只需几秒钟。[1]

2.4 节详细介绍了设置 DigitalOcean Droplet 的方法，你可以调整其中的 bash 脚本，以反映对 Python 库的个别需求。

操作风险

虽然自动化算法交易策略的开发与测试可以在本地计算机（台式电脑、笔记本电脑等）上进行，但这种环境不适于部署实盘交易策略。Web 连接丢失或者短暂的停电可能使整个算法瘫痪，导致投资组合里出现意外的开仓头寸，或者导致数据集损坏（由于遗漏了实时分笔交易数据），可能产生错误信号和意外的交易/头寸。

16.5 日志与监控

假定自动化算法交易策略部署在一台远程服务器（云实例、租赁服务器等）上，所有必要的 Python 库已经安装（参见 2.4 节），而且 Jupyter Notebook 也已经安全地运行。从算法交易者的角度看，如果他们不希望整天坐在屏幕前面和登录到服务器上，还需要考虑什么呢？

本节介绍这方面的两个重要主题：日志与实时监控。日志在磁盘上保存信息和事件以供以后检查，这是软件应用开发与部署中的标准做法。但是，我们的焦点应该在金融方面，要记录重要的金融数据与时间信息，供以后检查分析。使用套接字通信的实时监控也是如此。通过套接字，我们可以创建稳定、实时的重要金融特征流，即便部署发生在云端，我们也可以从本地计算机上读取和处理这些信息。

16.8.1 节介绍了一个实现以上特征的脚本，它利用了 16.4 节中的代码。该脚本中的代码可以根据持久化算法对象，在远程服务器上部署算法交易策略。它还根据一个自定义函数来添加日志与监控功能，该函数使用 ZeroMQ 进行套接字通信。与 16.8.2 节中的简短脚本相结合，它可以实施监控远程服务器上的活动。

1　使用邀请链接可以在 DigitalOcean 注册新账户时得到 10 美元的优惠。——原注

16.8.1 节中的脚本运行（本地或者远程）时，记录并通过套接字发送的输出如下所示：

```
2018-07-25 09:16:15.568208
=================================================================
NUMBER OF BARS: 24

=================================================================
MOST RECENT DATA
                          Mid   Returns  Direction
2018-07-25 07:15:30  1.168885 -0.000009         -1
2018-07-25 07:15:45  1.168945  0.000043          1
2018-07-25 07:16:00  1.168895 -0.000051         -1
2018-07-25 07:16:15  1.168895 -0.000009         -1
2018-07-25 07:16:30  1.168885 -0.000017         -1

=================================================================
features: [[ 1 -1 1 -1 -1]]
position: -1
signal:   -1


2018-07-25 09:16:15.581453
=================================================================
no trade placed

****END OF CYCLE***
2018-07-25 09:16:30.069737
=================================================================
NUMBER OF BARS: 25

=================================================================
MOST RECENT DATA
                          Mid   Returns  Direction
2018-07-25 07:15:45  1.168945  0.000043          1
2018-07-25 07:16:00  1.168895 -0.000051         -1
2018-07-25 07:16:15  1.168895 -0.000009         -1
2018-07-25 07:16:30  1.168950  0.000034          1
2018-07-25 07:16:45  1.168945 -0.000017         -1

=================================================================
features: [[-1 1 -1 -1 1]]
position: -1
signal:   1
```

```
2018-07-25 09:16:33.035094
=================================================================


=================================================
Going LONG.

    tradeId  amountK currency  grossPL  isBuy
0  61476318      100  EUR/USD       -2  True
=================================================


****END OF CYCLE***
```

在本地运行 16.8.2 节的脚本，可以实时读取和处理这些信息。当然，按照需求调整日志和流数据很容易。[1]同样，我们也可以持久化执行交易脚本期间创建的 `DataFrame` 对象。而且，交易脚本和整个逻辑都可以调整，以包含用程序方式止损或者采用利润目标的元素。我们还可以使用 FXCM 交易 API 中更复杂的订单类型。

考虑所有风险

交易货币组合和 CFD 涉及许多金融风险。为这些金融工具实施算法交易策略自然会导致一系列附加风险。其中包括交易与执行逻辑的缺陷，以及一些技术风险，如套接字通信问题、部署期间分笔数据读取延迟甚至丢失。因此，以自动化方式部署交易策略时，应该确保能识别、评估和处理所有相关市场、执行、运营、技术及其他风险。本章介绍的代码仅用于技术说明目的。

16.6　结语

本章介绍了算法交易策略——以机器学习分类算法为基础，预测市场变动方向的自动化部署。它包括各种重要主题：资本管理（以凯利标准为基础）、向量化绩效与风险事后检验、将离线算法转换为在线交易算法、对应基础设施部署，以及部署期间的日志与监控。

本章的主题很复杂，需要机器学习从业者具有广泛技能。另一方面，拥有来自 FXCM 等交易商的算法交易 REST API，可以明显简化自动化任务，因为其核心部分主要归结为使用 Python 包装器库 fxcmpy 读取数据和下单。围绕这一核心，我们还补充了缓解

1　注意，两个脚本中实现的套接字通信没有加密。在 Web 上发送明文，这在生产环境下可能带来安全风险。——原注

运营与技术风险的内容。

16.7　Python 脚本

16.7.1　自动化交易策略

下面是以自动化方式实施算法交易策略的 Python 脚本，它包括日志与监控：

```
#
# Automated ML-Based Trading Strategy for FXCM
# Online Algorithm, Logging, Monitoring
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import zmq
import time
import pickle
import fxcmpy
import numpy as np
import pandas as pd
import datetime as dt

sel = ['tradeId', 'amountK', 'currency',
       'grossPL', 'isBuy']

log_file = 'automated_strategy.log'

# loads the persisted algorithm object
algorithm = pickle.load(open('algorithm.pkl', 'rb'))

# sets up the socket communication via ZeroMQ (here: "publisher")
context = zmq.Context()
socket = context.socket(zmq.PUB)

# this binds the socket communication to all IP addresses of the machine
socket.bind('tcp://0.0.0.0:5555')


def logger_monitor(message, time=True, sep=True):
    ''' Custom logger and monitor function.
    '''
    with open(log_file, 'a') as f:
        t = str(dt.datetime.now())
```

```
        msg = ''
        if time:
            msg += '\n' + t + '\n'
        if sep:
            msg += 66 * '=' + '\n'
        msg += message + '\n\n'
        # sends the message via the socket
        socket.send_string(msg)
        # writes the message to the log file
        f.write(msg)


def report_positions(pos):
    ''' Prints, logs and sends position data.
    '''
    out = '\n\n' + 50 * '=' + '\n'
    out += 'Going {}.\n'.format(pos) + '\n'
    time.sleep(2) # waits for the order to be executed
    out += str(api.get_open_positions()[sel]) + '\n'
    out += 50 * '=' + '\n'
    logger_monitor(out)
    print(out)


def automated_strategy(data, dataframe):
    ''' Callback function embodying the trading logic.
    '''
    global min_bars, position, df
    # resampling of the tick data
    df = dataframe.resample(bar, label='right').last().ffill()

    if len(df) > min_bars:
        min_bars = len(df)
        logger_monitor('NUMBER OF TICKS: {} | '.format(len(dataframe)) +
                       'NUMBER OF BARS: {}'.format(min_bars))
        # data processing and feature preparation
        df['Mid'] = df[['Bid', 'Ask']].mean(axis=1)
        df['Returns'] = np.log(df['Mid'] / df['Mid'].shift(1))
        df['Direction'] = np.where(df['Returns'] > 0, 1, -1)
        # picks relevant points
        features = df['Direction'].iloc[-(lags + 1):-1]
        # necessary reshaping
        features = features.values.reshape(1, -1)
        # generates the signal (+1 or -1)
        signal = algorithm.predict(features)[0]
```

```
        # logs and sends major financial information
        logger_monitor('MOST RECENT DATA\n' +
                       str(df[['Mid', 'Returns', 'Direction']].tail()),
                       False)
        logger_monitor('features: ' + str(features) + '\n' +
                       'position: ' + str(position) + '\n' +
                       'signal:   ' + str(signal), False)

        # trading logic
        if position in [0, -1] and signal == 1: # going long?
            api.create_market_buy_order(
                symbol, size - position * size) # places a buy order
            position = 1 # changes position to long
            report_positions('LONG')

        elif position in [0, 1] and signal == -1: # going short?
            api.create_market_sell_order(
                symbol, size + position * size) # places a sell order
            position = -1 # changes position to short
            report_positions('SHORT')
        else: # no trade
           logger_monitor('no trade placed')

        logger_monitor('****END OF CYCLE***\n\n', False, False)

    if len(dataframe) > 350: # stopping condition
        api.unsubscribe_market_data('EUR/USD') # unsubscribes from data stream
        report_positions('CLOSE OUT')
        api.close_all() # closes all open positions
        logger_monitor('***CLOSING OUT ALL POSITIONS***')


if __name__ == '__main__':
    symbol = 'EUR/USD' # symbol to be traded
    bar = '15s' # bar length; adjust for testing and deployment
    size = 100 # position size in thousand currency units
    position = 0 # initial position
    lags = 5 # number of lags for features data
    min_bars = lags + 1 # minimum length for resampled DataFrame
    df = pd.DataFrame()
    # adjust configuration file location
    api = fxcmpy.fxcmpy(config_file='../fxcm.cfg')
    # the main asynchronous loop using the callback function
    api.subscribe_market_data(symbol, (automated_strategy,))
```

16.7.2　策略监控

下面的 Python 脚本通过套接字通信，实现自动化算法交易策略的本地或远程监控：

```
#
# Automated ML-Based Trading Strategy for FXCM
# Strategy Monitoring via Socket Communication
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import zmq

# sets up the socket communication via ZeroMQ (here: "subscriber")
context = zmq.Context()
socket = context.socket(zmq.SUB)

# adjust the IP address to reflect the remote location
socket.connect('tcp://REMOTE_IP_ADDRESS:5555')

# configures the socket to retrieve every message
socket.setsockopt_string(zmq.SUBSCRIBE, '')

while True:
    msg = socket.recv_string()
    print(msg)
```

16.8　延伸阅读

本章引用的论文如下：

- Rotando, Louis, and Edward Thorp (1992).“The Kelly Criterion and the Stock Market.” *The American Mathematical Monthly*, Vol. 99, No. 10, pp. 922-931.
- Hung, Jane (2010):“Betting with the Kelly Criterion.”

涵盖 Python 算法交易的全面在线培训课程请参见 Python Quants 官网。

第 5 部分

衍生品分析

本部分关注基于规模较小、但是仍然很强大的蒙特卡洛模拟期权及衍生品定价的实际应用的开发[1]。最终目标是一组 Python 类——我们称之为衍生品分析（Derivatives AnalytiX，DX）*定价库*。它可以完成以下工作。

建模

为了折现目的建立短期利率模型；建立欧式和美式期权的模型，包括它们的标的风险因素以及相关的市场环境；建立更复杂的投资组合模型，包含多种期权和多种可能相关的标的风险因素。

模拟

根据几何布朗运动、跳跃扩散和平方根扩散来模拟风险因素；同时，连贯地模拟一些此类风险因素，不管它们是否相关。

估值

按照风险中立估值方法，估算任意收益的欧式和美式期权价值；以一致、全面的方式估算由这些期权组成的投资组合的价值。

风险管理

以独立于标的风险因素或者行权类型的方式，从数值上估算非常重要的“期权希腊字母”——也就是期权/衍生品的 Delta 和 Vega 值。

1　期权交易以及相关主题（如市场基础知识以及所谓的“期权希腊字母”在期权风险管理中的角色）的简介和全面概述，请参见 James Bittman (2009)的 *Trading Options as a Professional*（McGraw Hill，New York）。——原注

应用

使用该库以市场为基础（使用经过检验的 DAX30 指数模型）估算以 DAX30 股票指数为标的的非交易美式期权投资组合价值，并加以管理。

本部分介绍的材料依赖于 DX 分析库，该库是由本书作者和 Python Quants 股份有限公司开发和提供的，读者也可通过 Quants 平台获取。该库的完整版本可以进行复杂、多风险衍生物和交易账册的建模、定价及风险管理。

本部分包括以下几章。

- 第 17 章以理论和技术的形式介绍估值框架。从理论上说，资产定价基本定理和风险中立估值方法是核心。而在技术方面，这一章介绍用于风险中立折现和市场环境的 Python 类。
- 第 18 章关注基于几何布朗运动、跳跃扩散和平方根扩散过程的风险因素模拟；讨论 1 个通用类和 3 个特殊类。
- 第 19 章介绍根据单一潜在风险因素估算单一欧式或者美式衍生品价值的方法；主要的组成部分仍然是一个通用类和两个特殊类。通用类可以独立于期权类型估算 Delta 和 Vega 值。
- 第 20 章介绍基于多种（可能相关的）潜在风险因素，估算可能包含多种衍生品的复杂投资组合的价值的方法；本章还将介绍一个用于建立衍生品头寸模型的简单类和一个更复杂的、用于一致投资组合估值的类。
- 第 21 章利用其他章中开发的 DX 库，来估算 DAX30 股票指数美式看跌期权投资组合的价值并对其加以管理。

第 17 章

估值框架

复利是有史以来最大的数学发现。

——阿尔伯特 • 爱因斯坦

本章介绍开发 DX 库所需的基本概念，并为其提供一个框架。我们将简短地回顾一下资产定价基本定理，这一理论为模拟和估值提供了理论背景。然后，本章介绍日期处理和风险中立折现。我们只考虑简单的情况——固定短期利率折现，更复杂和现实的模型很容易添加到库中。本章还介绍市场环境的概念——这是后续章节中几乎所有其他类实例化所需的一组常量、列表和曲线。

本章由以下内容组成。

资产定价基本定理

本节介绍资产定价基本定理，它为我们要开发的库提供了理论背景。

风险中立折现

本节开发一个类，该类用于期权与其他衍生工具未来收益的风险中立折现。

市场环境

本节开发一个类，该类可以管理单一金融工具与多金融工具投资组合定价的市场环境。

17.1 资产定价基本定理

资产定价基本定理是现代金融理论和数学的基石和成功案例之一[1]。它的中心思想是鞅

1 对所涉数学机制的全面研究和细节参见 Delbae 和 Schachermayer（2004）的著作。Hilpisch（2015）著作的第 4 章中对此也有简短的介绍，特别是离散时间的版本。——原注

（Martingale）测度——也就是从折现后风险因素（随机过程）中消去漂移的概率测度。换句话说，在鞅测度下，所有风险因素随无风险短期利率漂移——而不随包含某种无风险短期利率之上的风险溢价的任何其他市场利率漂移。

17.1.1 简单示例

考虑只有今天和明天两个交易日的一个简单经济环境，其中包含一种高风险资产（某种股票）和一种无风险资产（债券）。今天的债券价格为 10 美元，明天清偿价值为 10 美元（0 利率）。股票今天价格为 10 美元，明日售出价格为 20 美元和 0 美元的概率分别为 60%和 40%。债券的无风险收益为 0。股票的预期收益率为 $\frac{0.6\times 20+0.4\times 0}{10}-1=0.2$ 。这就是补偿股票风险特性的风险溢价。

现在考虑行权价为 15 美元的看涨期权。这种有 60%的概率取得 5 美元收益，在其他情况下收益为 0 的未定权益，其公允价值是多大呢？例如，我们可以取得期望值，并将结果值折现（这里使用的是 0 利率）。这种方法得到的价值是 0.6 × 5 = 3 美元，因为在股票价格上升至 20 美元时期权可以获得 5 美元的收益，其他情况下收益为 0。

但是，还有另一种方法已经成功地应用到期权定价问题上，问题为通过可交易证券的组合来复制期权收益。这很容易验证，购买 0.25 份股票就可以完全复制期权的收益（在 60%的情况下，我们的收益为 0.25 × 20 = 5 美元）。四分之一的股票成本仅为 2.5 美元而非 3 美元，在现实世界概率测度下的预期会过高估计期权的价值。

为什么会出现这种情况？现实世界的测度隐含着 20%的股票风险溢价，是因为股票中隐含的风险（盈利 100%或者损失 100%）"确实"无法依靠多样化或者对冲来消除。另一方面，有某种投资组合可以没有任何风险地复制期权的收益。这也意味着，卖出这种期权可以完全对冲任何风险[1]。这种由期权和对冲头寸组成的完全对冲投资组合必须得到等于无风险利率的收益，以避免套利机会（即"空手套白狼"，且概率为正数的机会）。

我们能否不通过预期估计看涨期权的价值？是的，可以做到。我们只需要改变概率，使风险资产（股票）随无风险利率（0）漂移即可。很显然，为两种情形同样设定 50%的（鞅）测度可以实现这一点；计算式为 $\frac{0.5\times 20+0.5\times 0}{10}-1=0$ 。现在，在新的鞅测度下计算期权收益，可以得到正确的（无套利）公允价值：0.5 × 5 + 0.5 × 0 = 2.5 美元。

17.1.2 一般结果

上述方法的迷人之处在于，它甚至可以用于非常复杂的经济环境，例如连续时间模型

1 这种策略包括以 2.5 美元购入期权，以 2.5 美元购买 0.25 股股票。这种组合不管简单经济中出现什么情形，其收益均为 0。——原注

（考虑连续的时间点）、大量风险资产、复杂的衍生品收益等。

因此，我们考虑离散时间中的一般市场模型[1]。

离散时间中的一般市场模型 μ 是以下元素的集合：

- 有限状态空间 Ω；
- 过滤 F；
- 在 $\wp(\Omega)$ 上定义的严格正概率测度 P；
- 终止日期 $T \in N$，$T < \infty$；
- K+1 个严格正向证券价格过程的集合 $S \equiv \{(S_t^k)_{t \in \{0,\cdots,T\}} : k \in \{0,\cdots,K\}\}$。

我们将模型写作 μ={（ Ω，$\wp(\Omega)$，F，P ），T，S}。

根据上述一般市场模型，我们可以这样描述资产定价基本定理[2]。

考虑一般市场模型 μ，根据资产定价基本定理，下面 3 个陈述等价：

- 市场模型 μ 中没有套利机会；
- P 等价鞅测度集合 Q 不为空；
- 一致线性定价系统集合 P 不为空。

对于未定权益（期权、衍生品、期货、远期、掉期等）的估值与定价，该定理的重要性可以用如下的推论阐述：

> 如果市场模型 μ 是无套利的，则对任何可获得（可复制）的未定权益（期权、衍生品等）V_T 存在一个相关的唯一价格 V_0，满足 $\forall Q \in Q : V_0 = \mathrm{E}_0^Q(\mathrm{e}^{-rT}V_T)$，其中 e^{-rT} 是固定短期利率 r 对应的风险中立折现因子。

这一结果说明了该定理的重要性，并说明了我们上面介绍的简单推论适用于一般市场模型。

由于鞅测度的作用，这种估值方法也常常被称作鞅方法，或者，因为鞅测度下所有风险资产都随无风险利率漂移而被称作风险中立估值方法。对于我们的目的来说，第二个术语可能更好，因为在数值应用中，我们“简单”地让风险因素（随机过程）根据风险中立短期利率漂移。对于我们的应用，没有必要处理概率测度。但是，这些测度从理论上证明了我们所应用的核心理论结果，以及实施的技术方法。

最后，考虑一般市场模型中的市场完全性：

> 如果市场 μ 是无套利的，且每种未定权益（期权、衍生品等）都可获得（可复制），则称它是完全的。

1　概率论概念参见 Williams（1991）。——原注

2　参见 Delbaen 和 Schachermayer（2004）。——原注

假定市场模型 μ 是无套利的，当且仅当 Q 是单例（Singleton，即存在唯一的 P 等价鞅测度）时称该市场是完全的。

本节完成了对后续小节所需的技术背景的讨论。对这些概念、定义和结果的详细阐述参考 Hilpisch（2015）的第 4 章。

17.2 风险中立折现

很明显，风险中立折现是风险中立估值方法的核心。首先需要开发一个用于风险中立折现的 Python 类。但在此之前，仔细地观察一下估值相关日期的建模和处理很有必要。

17.2.1 日期建模与处理

折现的必要前提之一是建立日期模型（参见附录 A）。出于估值的目的，通常将当日和一般市场模型最后日期 T 之间的一段时期分为离散的时间间隔。这些时间间隔可以是同构的（等长），也可以是异构的（不等长）。估值库应该能够处理异构时间间隔的一般情况，因为更简单的情况会自动包含。因此，我们使用日期列表，假定最小的相关时间间隔为一天。这意味着，我们不关心日间事件——如果需要处理这些事件，就必须建立时间模型（除了日期之外）[1]。

要编辑相关日期列表，基本上可以采用两种方法：构建一个离散的日期列表（例如 Python 中的 `datetime` 对象）或者年分数（十进制数字，就像在理论工作中常常做的那样）。

首先是导入：

```
In [1]: import numpy as np
        import pandas as pd
        import datetime as dt

In [2]: from pylab import mpl, plt
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline

In [3]: import sys
        sys.path.append('../dx')
```

例如，下面的 `dates` 和 `fraction` 定义（大致）等价：

```
In [4]: dates = [dt.datetime(2020, 1, 1), dt.datetime(2020, 7, 1),
                 dt.datetime(2021, 1, 1)]
```

1 添加时间成分实际上很简单，但是为了易于讲解，在这里不这么做。——原注

```
In [5]: (dates[1] - dates[0]).days / 365.
Out[5]: 0.4986301369863014

In [6]: (dates[2] - dates[1]).days / 365.
Out[6]: 0.5041095890410959

In [7]: fractions = [0.0, 0.5, 1.0]
```

它们只是大致等价，因为年分数很少处于某一天的开始（0 点）。你只需要想想，将一年除以 50 的结果就明白了。

有时候，根据日期列表得出年分数是必要的。`get_year_delatas()`函数可以完成这项工作。

```
#
# DX Package
#
# Frame -- Helper Function
#
# get_year_deltas.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

def get_year_deltas(date_list, day_count=365.):
    ''' Return vector of floats with day deltas in year fractions.
    Initial value normalized to zero.

    Parameters
    ==========
    date_list: list or array
        collection of datetime objects
    day_count: float
        number of days for a year
        (to account for different conventions)

    Results
    =======
    delta_list: array
        year fractions
    '''

    start = date_list[0]
```

```
    delta_list = [(date - start).days / day_count
                  for date in date_list]
    return np.array(delta_list)
```

这个函数的应用如下：

```
In [8]: from get_year_deltas import get_year_deltas

In [9]: get_year_deltas(dates)
Out[9]: array([0.        , 0.49863014, 1.00273973])
```

在建立短期利率模型时，这个函数的好处显而易见。

17.2.2 恒定短期利率

我们将焦点放在短期利率折现的最简单情况——也就是短期利率一直保持不变的情况。许多期权定价模型，如 Black-Scholes-Merton（1973）、Merton（1976）或 Cox-Ross-Rubinstein（1979）都做出这个假设[1]。我们假定连续折现，因为是期权定价应用中常见的假设。在这种情况下，在给定未来日期 t 和固定短期利率 r 下，今天的一般折现因子可以由 $D_0(t) = e^{-rt}$ 得出。当然，在结束日有一个特例 $D_0(T) = e^{-rT}$。注意，t 和 T 都是年分数。

折现因子也可以解释为分别在 t 和 T 到期的单位零息债券（ZCB）的当日价值[2]。给定两个日期 $t \geqslant s \geqslant 0$，从 t 到 s 的折现因子可以由公式 $D_s(t) = D_0(t)/D_0(s) = e^{-rt}/e^{-rs} = e^{-rt} \cdot e^{rs} = e^{-r(t-s)}$ 给出。

下面将上述考虑因素转换为 Python 代码的一个 Python 类[3]：

```
#
# DX Library
#
# Frame -- Constant Short Rate Class
#
# constant_short_rate.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
from get_year_deltas import *
```

1 例如，对于短期期权的定价，这个假设在许多情况下都满足。——原注

2 单位零息债券在到期时偿付一个货币单位，在当日和到期日之间没有利息。——原注

3 Python 面向对象开发的基本知识参见第 13 章。在本章和本节的余下内容中，我们从标准的 PEP8 命名约定中得出 Python 类名。PEP 8 推荐为 Python 类名使用“词首字母大写”或者“驼峰式大小写”。在“接口已经记入文档并主要作为可调用对象的情况下”使用 PEP 8 提到的函数命名惯例作为有效的替代方案。——原注

```
class constant_short_rate(object):
    ''' Class for constant short rate discounting.

    Attributes
    ==========
    name: string
        name of the object
    short_rate: float (positive)
        constant rate for discounting

    Methods
    =======
    get_discount_factors:
        get discount factors given a list/array of datetime objects
        or year fractions
    '''

    def __init__(self, name, short_rate):
        self.name = name
        self.short_rate = short_rate
        if short_rate < 0:
            raise ValueError('Short rate negative.')
            # this is debatable given recent market realities

    def get_discount_factors(self, date_list, dtobjects=True):
        if dtobjects is True:
            dlist = get_year_deltas(date_list)
        else:
            dlist = np.array(date_list)
        dflist = np.exp(self.short_rate * np.sort(-dlist))
        return np.array((date_list, dflist)).T
```

dx.constant_short_rate 类的应用最好由一个简单、具体的例子说明。主要的结果是一个二维 ndarray 对象，它包含成对的 datetime 对象和相关折现因子。该类（特别是对象 csr）通常也能处理年分数：

```
In [10]: from constant_short_rate import constant_short_rate

In [11]: csr = constant_short_rate('csr', 0.05)

In [12]: csr.get_discount_factors(dates)
Out[12]: array([[datetime.datetime(2020, 1, 1, 0, 0), 0.9510991280247174],
               [datetime.datetime(2020, 7, 1, 0, 0), 0.9753767163648953],
               [datetime.datetime(2021, 1, 1, 0, 0), 1.0]], dtype=object)
```

```
In [13]: deltas = get_year_deltas(dates)
         deltas
Out[13]: array([0.        , 0.49863014, 1.00273973])

In [14]: csr.get_discount_factors(deltas, dtobjects=False)
Out[14]: array([[0.        , 0.95109913],
                [0.49863014, 0.97537672],
                [1.00273973, 1.        ]])
```

这个类负责其他类中需要的所有折现运算。

17.3 市场环境

市场环境“只是”其他数据和 Python 对象集合的一个名称。但是，使用这一抽象概念相当方便，因为它简化了许多操作，而且考虑到了循环特征的一致性建模[1]。市场环境主要由 3 个字典组成，它们存储以下类型的数据和 Python 对象。

常量

例如模型参数或者期权到期日。

列表

一般是对象序列，例如用于建立（风险）证券模型对象的列表对象。

曲线

用于折现的对象；例如，`dx.constant_short_rate` 类的实例。

下面是 `dx.market_environment` 类的代码。字典对象的处理参见第 3 章。

```
#
# DX Package
#
# Frame -- Market Environment Class
#
# market_environment.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#

class market_environment(object):
    ''' Class to model a market environment relevant for valuation.
```

1 这个概念的相关信息参见 Fletcher 和 Gardner（2009），他们大量使用了市场环境。——原注

```
    Attributes
    ==========
    name: string
        name of the market environment
    pricing_date: datetime object
        date of the market environment

    Methods
    =======
    add_constant:
        adds a constant (e.g. model parameter)
    get_constant:
        gets a constant
    add_list:
        adds a list (e.g. underlyings)
    get_list:
        gets a list
    add_curve:
        adds a market curve (e.g. yield curve)
    get_curve:
        gets a market curve
    add_environment:
        adds and overwrites whole market environments
        with constants, lists, and curves
    '''

    def __init__(self, name, pricing_date):
        self.name = name
        self.pricing_date = pricing_date
        self.constants = {}
        self.lists = {}
        self.curves = {}

    def add_constant(self, key, constant):
        self.constants[key] = constant

    def get_constant(self, key):
        return self.constants[key]

    def add_list(self, key, list_object):
        self.lists[key] = list_object

    def get_list(self, key):
        return self.lists[key]
```

```
    def add_curve(self, key, curve):
        self.curves[key] = curve

    def get_curve(self, key):
        return self.curves[key]

    def add_environment(self, env):
        # overwrites existing values, if they exist
        self.constants.update(env.constants)
        self.lists.update(env.lists)
        self.curves.update(env.curves)
```

虽然 dx.market_environment 类没有什么特殊的，但是一个简单的例子就可以说明使用该类的实例有多么方便：

```
In [15]: from market_environment import market_environment

In [16]: me = market_environment('me_gbm', dt.datetime(2020, 1, 1))

In [17]: me.add_constant('initial_value', 36.)

In [18]: me.add_constant('volatility', 0.2)

In [19]: me.add_constant('final_date', dt.datetime(2020, 12, 31))

In [20]: me.add_constant('currency', 'EUR')

In [21]: me.add_constant('frequency', 'M')

In [22]: me.add_constant('paths', 10000)

In [23]: me.add_curve('discount_curve', csr)

In [24]: me.get_constant('volatility')
Out[24]: 0.2

In [25]: me.get_curve('discount_curve').short_rate
Out[25]: 0.05
```

以上的例子说明了对这个相当通用的“存储”类的基本处理。对于实用的应用程序，首先收集市场数据和其他数据以及 Python 对象，然后实例化 dx.market_environment 对象，并填入相关数据及对象。只用一步就可以将其交付给需要保存在对应 dx.market_environment 对象中的其他类。

这种面向对象建模方法的重大优势之一是，如 dx.constant_short_rate 类的实例

可以存在于多个环境中。一旦实例被更新（如设置新的固定短期利率），包含这个特殊折现类的所有 `dx.market_environment` 实例都将被自动更新。

> **灵活性**
>
> 本节介绍的市场环境类是一种灵活的手段，可以为与期权、衍生品及由其组成的投资组合定价相关的数量和输入数据建模并提供存储。但是，这种灵活性也带来了操作风险，很容易在实例化时传递无意义的数据、对象等，实例化也许能够捕捉到它们，也可能捕捉不到。在生产环境中，必须增加一些检查，至少捕捉到明显错误的情况。

17.4　结语

本章为构建 Python 库、通过蒙特卡洛模拟估算期权和其他衍生品价值的较大项目提供一个框架。本章介绍资产定价基本定理，用相当简单的数值化示例加以阐述。在这一点上，本章为离散时间的一般市场模型提供了重要成果。

本章还为风险中立折现开发了一个 Python 类，以数值形式运用了资产定价基本定理的机制。根据 Python `datetime` 对象列表或者表示年分数的浮点对象，`dx.constant_short_rate` 类的实例提供对应的折现因子（单位零息债券的现值）。

本章的最后介绍了相当通用的 `dx.market_environment` 类，该类集合了用于建模、模拟、估值和其他目的的相关数据和 Python 对象。

为了简化未来的导入工作，我们将使用一个包装器模块 `dx_frame.py`：

```
#
# DX Analytics Package
#
# Frame Functions & Classes
#
# dx_frame.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import datetime as dt

from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment
```

下面的一条 `import` 语句可以仅通过一步使所有框架组件可用：

```
import dx_frame
```

对于 Python 库和模块套件，还有一个选择是将所有相关 Python 模块保存在一个（子）目录中，并在该目录中放入一个特殊的`__init__.py` 文件进行所有导入工作。例如，将所有模块保存在 `dx` 子目录下，例 15-5 中提供的文件就可以完成导入。不过，要注意这个特殊文件的命名约定。

```
#
# DX Package
# packaging file
# __init__.py
#
import datetime as dt

from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment
```

在这种情况下，只要使用目录名就可以一次完成所有导入：

```
from dx import *
```

还可以使用如下方法：

```
import dx
```

17.5 延伸阅读

对于本章介绍的主题，下面是一些有用的图书资源：

- Bittman, James (2009). *Trading Options as a Professional*. New York: McGraw Hill.
- Delbaen, Freddy, and Walter Schachermayer (2004). *The Mathematics of Arbitrage*. Berlin, Heidelberg: Springer-Verlag.
- Fletcher, Shayne, and Christopher Gardner (2009). *Financial Modelling in Python*. Chichester, England: Wiley Finance.
- Hilpisch, Yves (2015). *Derivatives Analytics with Python*. Chichester, England: Wiley Finance.
- Williams, David (1991). *Probability with Martingales*. Cambridge, England: Cambridge University Press.

定义本章中引用模型的原创研究论文，参考后续章的“延伸阅读”部分。

第 18 章

金融模型的模拟

科学的目的不是分析或者描述，而是制作这个世界的实用模型。

——爱德华·德·波诺

第 12 章介绍了使用 Python 和 NumPy 实现随机过程蒙特卡洛模拟的一些细节。本章应用前面介绍的基本技术实现模拟类，并将其作为 DX 库的核心组件。我们将注意力限制在 3 个广泛使用的随机过程上。本章包含如下内容：

随机数生成

本节开发一个函数，用方差缩减技术生成标准正态分布随机数。[1]

通用模拟类

本节开发一个通用模拟类，其他特定模拟类将从该类继承基本属性与方法。

几何布朗运动

本节介绍几何布朗运动，它是由 Black 和 Scholes（1973）的开创性工作引入期权定价文献的。几何布朗运动在本书中被多次使用，尽管有一些明显的不足之处，而且这些不足在金融现实中也得到了越来越多的证实，但它仍是期权和衍生品估值的一个基准过程。

跳跃扩散

跳跃扩散由 Merton（1976）引入金融，它在几何布朗运动（GBM）上增加了对数分布跳跃成分；这使我们可以考虑更多情况，例如短期价外（OTM）期权的定价往往需要考虑大幅暴涨的可能性。换言之，以 GBM 作为金融模型往往无法很好地解释这种 OTM 期权的市场价值，而跳跃扩散可以做到。

1　我们知道，文中所说的“随机”数通常只是“伪随机数”。——原注

平方根扩散

> 平方根扩散在金融学中的流行原因是 Cox、Ingersoll 和 Ross（1985）的努力，它被用于建立利率和波动率等均值回归数量的模型。除了均值回归之外，平方根扩散过程总是正向的，这通常是理想特性。

本章介绍的模型模拟的进一步细节参见 Hilpisch（2015），这本书还特别包含了一个基于 Merton 跳跃扩散（1976）的完整案例研究。

18.1 随机数生成

随机数生成是蒙特卡洛模拟的核心任务之一[1]。第 12 章说明了如何使用 Python 和 `numpy.random` 等库生成不同分布的随机数。对于我们手上的项目，标准正态分布随机数尤为重要。因此，创建一个方便的函数来生成这种特殊类型的随机数是值得的，函数如下面定义的 `sn_random_numbers()`：

```
#
# DX Package
#
# Frame -- Random Number Generation
#
# sn_random_numbers.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np


def sn_random_numbers(shape, antithetic=True, moment_matching=True,
                      fixed_seed=False):
    ''' Returns an ndarray object of shape shape with (pseudo)random numbers
    that are standard normally distributed.
    Parameters
    ==========
    shape: tuple (o, n, m)
        generation of array with shape (o, n, m)
    antithetic: Boolean
        generation of antithetic variates
    moment_matching: Boolean
        matching of first and second moments
```

1 随机数和随机变量生成的相关信息参见 Glasserman（2004）第 2 章。——原注

```
    fixed_seed: Boolean
        flag to fix the seed

    Results
    =======
    ran: (o, n, m) array of (pseudo)random numbers
    '''
    if fixed_seed:
        np.random.seed(1000)
    if antithetic:
        ran = np.random.standard_normal(
            (shape[0], shape[1], shape[2] // 2))
        ran = np.concatenate((ran, -ran), axis=2)
    else:
        ran = np.random.standard_normal(shape)
    if moment_matching:
        ran = ran - np.mean(ran)
        ran = ran / np.std(ran)
    if shape[0] == 1:
        return ran[0]
    else:
        return ran
```

本函数中使用的方差缩减技术，即反向路径和矩匹配，在第 12 章中也有介绍[1]。`sn_random_numbers()`函数的应用很简单：

```
In [26]: from sn_random_numbers import *

In [27]: snrn = sn_random_numbers((2, 2, 2), antithetic=False,
                                  moment_matching=False, fixed_seed=True)
         snrn
Out[27]: array([[[-0.8044583 , 0.32093155],
                 [-0.02548288, 0.64432383]],

                [[-0.30079667, 0.38947455],
                 [-0.1074373 , -0.47998308]]])

In [28]: round(snrn.mean(), 6)
Out[28]: -0.045429

In [29]: round(snrn.std(), 6)
Out[29]: 0.451876

In [30]: snrn = sn_random_numbers((2, 2, 2), antithetic=False,
```

1 Glasserman(2004)的第 4 章提供了不同方差缩减技术的概述和理论细节。——原注

```
                                   moment_matching=True, fixed_seed=True)
         snrn
Out[30]: array([[[-1.67972865, 0.81075283],
                 [ 0.04413963, 1.52641815]],

                [[-0.56512826, 0.96243813],
                 [-0.13722505,-0.96166678]]])

In [31]: round(snrn.mean(), 6)
Out[31]: -0.0

In [32]: round(snrn.std(), 6)
Out[32]: 1.0
```

这个函数将成为后面的模拟类中的“主力干将”。

18.2　通用模拟类

面向对象建模（正如第 6 章中的介绍）允许属性和方法的继承。这是我们在构建模拟类时想要利用的：从一个通用模拟类开始，该类包含所有其他模拟类共享的属性和方法，然后可以将焦点放在被模拟随机过程的特定元素相关的其他类上。通过提供以下 3 个属性可实例化任何模拟类的对象。

name

　　用作模型模拟对象名称的字符串对象。

mar_env

　　`dx.market_environment` 类的一个实例。

corr

　　表示对象是否相关的一个标志（布尔型）。

这再次说明了市场环境的作用：一次性提供模拟和估值所需的所有数据和对象。通用类的方法如下。

generate_time_grid()

　　这个方法生成模拟所用的相关日期的“时间网格”。此任务对于每个模拟类都相同。

get_instrument_values()

　　每个模拟类都必须返回包含模拟金融工具价值的 `ndarray` 对象（例如模拟的股票价格、商品价格、波动率）。

下面是通用模型模拟类的代码。其中的方法利用了模型定制类提供的其他方法，如 `self.generate_paths()`。在我们对专用、非通用模拟类有全面的认识之后，这方面的细节就会变得很清晰。

```
#
# DX Package
#
# Simulation Class -- Base Class
#
# simulation_class.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
import pandas as pd


class simulation_class(object):
    ''' Providing base methods for simulation classes.

    Attributes
    ==========
    name: str
        name of the object
    mar_env: instance of market_environment
        market environment data for simulation
    corr: bool
        True if correlated with other model object

    Methods
    =======
    generate_time_grid:
        returns time grid for simulation
    get_instrument_values:
        returns the current instrument values (array)
    '''

    def __init__(self, name, mar_env, corr):
        self.name = name
        self.pricing_date = mar_env.pricing_date
        self.initial_value = mar_env.get_constant('initial_value')
        self.volatility = mar_env.get_constant('volatility')
        self.final_date = mar_env.get_constant('final_date')
        self.currency = mar_env.get_constant('currency')
```

```
            self.frequency = mar_env.get_constant('frequency')
        self.paths = mar_env.get_constant('paths')
        self.discount_curve = mar_env.get_curve('discount_curve')
        try:
            # if time_grid in mar_env take that object
            # (for portfolio valuation)
            self.time_grid = mar_env.get_list('time_grid')
        except:
            self.time_grid = None
        try:
            # if there are special dates, then add these
            self.special_dates = mar_env.get_list('special_dates')
        except:
            self.special_dates = []
        self.instrument_values = None
        self.correlated = corr
        if corr is True:
            # only needed in a portfolio context when
            # risk factors are correlated
            self.cholesky_matrix = mar_env.get_list('cholesky_matrix')
            self.rn_set = mar_env.get_list('rn_set')[self.name]
            self.random_numbers = mar_env.get_list('random_numbers')

    def generate_time_grid(self):
        start = self.pricing_date
        end = self.final_date
        # pandas date_range function
        # freq = e.g. 'B' for Business Day,
        # 'W' for Weekly, 'M' for Monthly
        time_grid = pd.date_range(start=start, end=end,
                                  freq=self.frequency).to_pydatetime()
        time_grid = list(time_grid)
        # enhance time_grid by start, end, and special_dates
        if start not in time_grid:
            time_grid.insert(0, start)
            # insert start date if not in list
        if end not in time_grid:
            time_grid.append(end)
            # insert end date if not in list
        if len(self.special_dates) > 0:
            # add all special dates
            time_grid.extend(self.special_dates)
            # delete duplicates
            time_grid = list(set(time_grid))
            # sort list
```

```python
            time_grid.sort()
        self.time_grid = np.array(time_grid)

    def get_instrument_values(self, fixed_seed=True):
        if self.instrument_values is None:
            # only initiate simulation if there are no instrument values
            self.generate_paths(fixed_seed=fixed_seed, day_count=365.)
        elif fixed_seed is False:
            # also initiate resimulation when fixed_seed is False
            self.generate_paths(fixed_seed=fixed_seed, day_count=365.)
        return self.instrument_values
```

市场环境的解析被嵌入特殊方法`__init__`中，该方法在实例化时被调用。为了保持代码的简洁，我们不实施任何完整性检查。例如，不管内容是否真的是折现类的一个实例，下面的代码行都被认为是“成功”。因此，在编译和将`dx.market_environment`对象传递给任何模拟类时应该相当小心：

```python
    self.discount_curve = mar_env.get_curve('discount_curve')
```

表 18-1 展示了通用和其他所有模拟类的`dx.market_environment`对象必须包含的所有组成部分。

表 18-1　所有模拟类的市场环境元素

元　　素	类型	强制	描　　述
`initial_value`	常量	是	`pricing_date`（定价日）时的过程初始值
`volatility`	常量	是	过程的波动性系数
`final_date`	常量	是	模拟范围
`currency`	常量	是	金融实体的货币
`frequency`	常量	是	日期频率，和`pandas freq`参数相同
`paths`	常量	是	模拟路径数量
`discount_curve`	曲线	是	`dx.constant_short_rate`实例
`time_grid`	列表	否	相关日期的时间网格（在投资组合背景下）
`random_numbers`	列表	否	随机数数组（用于相关对象）
`cholesky_matrix`	列表	否	Cholesky 矩阵（用于相关对象）
`rn_set`	列表	否	包含指向相关随机数值指针的字典对象

与模型模拟对象关联有关的所有元素都将在后续的章节中解释。在本章中，我们将重点放在单一、无关联过程的模拟上。类似地，传递`time_grid`的选项只在投资组合背景下适用，这也将在后面解释。

18.3　几何布朗运动

几何布朗运动是公式 18-1 所描述的随机过程（参见第 12 章的公式 12-2，特别是参数和变量的含义）。过程的漂移已经被设置为无风险固定短期利率 r，这意味着我们在等价鞅测度下工作（参见第 17 章）。

公式 18-1　几何布朗运动的随机微分方程

$$dS_t = rS_t dt + \sigma S_t dZ_t$$

公式 18-2 提供了上述微分方程用于模拟目的的欧拉离散化格式（更多细节参见第 12 章的公式 12-3）。总体框架是离散时间市场模型（例如第 17 章中的一般市场模型 μ），使用有限相关日期集合 $0<t_1<t_2<\cdots<T$。

公式 18-2　模拟几何布朗运动的微分方程

$$S_{t_{m+1}} = S_{t_m} \exp\left(\left(r - \frac{1}{2}\sigma^2\right)(t_{m+1} - t_m) + \sigma\sqrt{t_{m+1} - t_m}\ z_t\right)$$
$$0 \leqslant t_m < t_{m+1} \leqslant T$$

18.3.1　模拟类

以下是 GBM 模型的一个专用类：

```
#
# DX Package
#
# Simulation Class -- Geometric Brownian Motion
#
# geometric_brownian_motion.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class


class geometric_brownian_motion(simulation_class):
    ''' Class to generate simulated paths based on
    the Black-Scholes-Merton geometric Brownian motion model.
```

```
    Attributes
    ==========
    name: string
        name of the object
    mar_env: instance of market_environment
        market environment data for simulation
    corr: Boolean
        True if correlated with other model simulation object

    Methods
    =======
    update:
        updates parameters
    generate_paths:
        returns Monte Carlo paths given the market environment
    '''

    def __init__(self, name, mar_env, corr=False):
        super(geometric_brownian_motion, self).__init__(name, mar_env, corr)

    def update(self, initial_value=None, volatility=None, final_date=None):
        if initial_value is not None:
            self.initial_value = initial_value
        if volatility is not None:
            self.volatility = volatility
        if final_date is not None:
            self.final_date = final_date
        self.instrument_values = None

    def generate_paths(self, fixed_seed=False, day_count=365.):
        if self.time_grid is None:
            # method from generic simulation class
            self.generate_time_grid()
        # number of dates for time grid
        M = len(self.time_grid)
        # number of paths
        I = self.paths
        # ndarray initialization for path simulation
        paths = np.zeros((M, I))
        # initialize first date with initial_value
        paths[0] = self.initial_value
        if not self.correlated:
            # if not correlated, generate random numbers
            rand = sn_random_numbers((1, M, I),
                                     fixed_seed=fixed_seed)
```

```
        else:
            # if correlated, use random number object as provided
            # in market environment
            rand = self.random_numbers
        short_rate = self.discount_curve.short_rate
        # get short rate for drift of process
        for t in range(1, len(self.time_grid)):
            # select the right time slice from the relevant
            # random number set
            if not self.correlated:
                ran = rand[t]
            else:
                ran = np.dot(self.cholesky_matrix, rand[:, t, :])
                ran = ran[self.rn_set]
            dt = (self.time_grid[t] - self.time_grid[t - 1]).days / day_count
            # difference between two dates as year fraction
            paths[t] = paths[t - 1] * np.exp((short_rate - 0.5 *
                                              self.volatility ** 2) *
dt +
                                              self.volatility * np.
sqrt(dt) * ran)
            # generate simulated values for the respective date
        self.instrument_values = paths
```

在这个特例中，`dx.market_environment` 对象只能包含表 18-1 中所示的数据和对象，即成分的最小集合。

`update()`方法的作用正如其名：它可以更新选择的模型重要参数。`generate_paths()` 方法更复杂一些，它有一些内嵌注释，可以清晰地说明非常重要的特征。理论上，考虑不同模型模拟对象之间的关联会给该方法带来一定的复杂度。考虑对象之间的关联的目的将会逐渐变得清晰，特别是在第 20 章中。

18.3.2　用例

下面的交互式 IPython 会话说明了几何布朗运动模拟类的使用方法。首先，我们必须生成包含所有强制性元素的 `dx.market_environment` 对象：

```
In [33]: from dx_frame import *

In [34]: me_gbm = market_environment('me_gbm', dt.datetime(2020, 1, 1))

In [35]: me_gbm.add_constant('initial_value', 36.)
         me_gbm.add_constant('volatility', 0.2)
         me_gbm.add_constant('final_date', dt.datetime(2020, 12, 31))
         me_gbm.add_constant('currency', 'EUR')
         me_gbm.add_constant('frequency', 'M') ❶
```

```
        me_gbm.add_constant('paths', 10000)

In [36]: csr = constant_short_rate('csr', 0.06)

In [37]: me_gbm.add_curve('discount_curve', csr)
```

❶ 每月频度，默认为月底。

其次，我们初始化一个模型模拟对象：

```
In [38]: from geometric_brownian_motion import geometric_brownian_motion

In [39]: gbm = geometric_brownian_motion('gbm', me_gbm) ❶

In [40]: gbm.generate_time_grid() ❷

In [41]: gbm.time_grid ❸
Out[41]: array([datetime.datetime(2020, 1, 1, 0, 0),
                datetime.datetime(2020, 1, 31, 0, 0),
                datetime.datetime(2020, 2, 29, 0, 0),
                datetime.datetime(2020, 3, 31, 0, 0),
                datetime.datetime(2020, 4, 30, 0, 0),
                datetime.datetime(2020, 5, 31, 0, 0),
                datetime.datetime(2020, 6, 30, 0, 0),
                datetime.datetime(2020, 7, 31, 0, 0),
                datetime.datetime(2020, 8, 31, 0, 0),
                datetime.datetime(2020, 9, 30, 0, 0),
                datetime.datetime(2020, 10, 31, 0, 0),
                datetime.datetime(2020, 11, 30, 0, 0),
                datetime.datetime(2020, 12, 31, 0, 0)], dtype=object)

In [42]: %time paths_1 = gbm.get_instrument_values() ❹
         CPU times: user 21.3 ms, sys: 6.74 ms, total: 28.1 ms
         Wall time: 40.3 ms

In [43]: paths_1.round(3) ❹
Out[43]: array([[36.   , 36.   , 36.   , ..., 36.   , 36.   , 36.   ],
                [37.403, 38.12 , 34.4  , ..., 36.252, 35.084, 39.668],
                [39.562, 42.335, 32.405, ..., 34.836, 33.637, 37.655],
                ...,
                [40.534, 33.506, 23.497, ..., 37.851, 30.122, 30.446],
                [42.527, 36.995, 21.885, ..., 36.014, 30.907, 30.712],
                [43.811, 37.876, 24.1  , ..., 36.263, 28.138, 29.038]])

In [44]: gbm.update(volatility=0.5) ❺

In [45]: %time paths_2 = gbm.get_instrument_values() ❺
```

```
CPU times: user 27.8 ms, sys: 3.91 ms, total: 31.7 ms
Wall time: 19.8 ms
```

❶ 实例化模拟对象。

❷ 生成时间网格。

❸ 显示时间网格，注意加入的初始日期。

❹ 按照参数模拟路径。

❺ 更新波动率参数并重复模拟。

图 18-1 展示了两组不同参数的 10 条模拟路径，很容易看出增大波动率参数值的效果：

```
In [46]: plt.figure(figsize=(10, 6))
         p1 = plt.plot(gbm.time_grid, paths_1[:, :10], 'b')
         p2 = plt.plot(gbm.time_grid, paths_2[:, :10], 'r-.')
         l1 = plt.legend([p1[0], p2[0]],
                         ['low volatility', 'high volatility'], loc=2)
         plt.gca().add_artist(l1)
         plt.xticks(rotation=30);
```

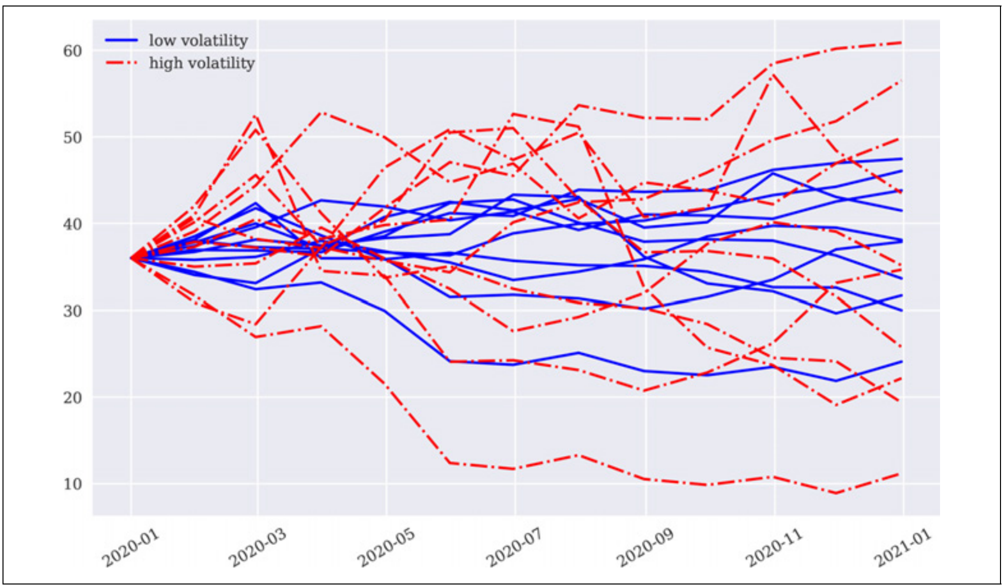


图 18-1　GBM 模拟类中的模拟路径



模拟的向量化

第 12 章中已经说明，NumPy 和 pandas 的向量化方法很适合编写简洁且高性能的模拟代码。

18.4 跳跃扩散

有了 `dx.geometric_brownian_motion` 类的背景知识后，实现 Merton（1976）描述的跳跃扩散模型就很简单了。跳跃扩散的随机微分方程如公式 18-3 所示（参见第 12 章中的公式 12-8，尤其是参数和变量的含义）。

公式 18-3 Merton 跳跃扩散模型的随机微分方程

$$dS_t = (r - r_J)S_t dt + \sigma S_t dZ_t + J_t S_t dN_t$$

公式 18-4 提供了用于模拟目的的欧拉离散化格式（更详细的解释参见第 12 章的公式 12-9）。

公式 18-4 Merton 跳跃扩散模型的欧拉离散化

$$S_{t_{m+1}} = S_{t_m}\left(\exp\left(\left(r - r_J - \frac{1}{2}\sigma^2\right)(t_{m+1} - t_m) + \sigma\sqrt{t_{m+1} - t_m}\, z_t^1\right) + (e^{\mu_J + \delta z_t^2} - 1)y_t\right)$$

$$0 \leqslant t_m < t_{m+1} \leqslant T$$

18.4.1 模拟类

下面是 `dx.jump_diffusion` 模拟类的 Python 代码。这个类现在所包含的内容都在意料之中。当然，模型不同，但是设计和方法本质上相同：

```
#
# DX Package
#
# Simulation Class -- Jump Diffusion
#
# jump_diffusion.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class

class jump_diffusion(simulation_class):
    ''' Class to generate simulated paths based on
    the Merton (1976) jump diffusion model.
```

```
    Attributes
    ==========
    name: str
        name of the object
    mar_env: instance of market_environment
        market environment data for simulation
    corr: bool
        True if correlated with other model object

    Methods
    =======
    update:
        updates parameters
    generate_paths:
    returns Monte Carlo paths given the market environment
'''

def __init__(self, name, mar_env, corr=False):
    super(jump_diffusion, self).__init__(name, mar_env, corr)
    # additional parameters needed
    self.lamb = mar_env.get_constant('lambda')
    self.mu = mar_env.get_constant('mu')
    self.delt = mar_env.get_constant('delta')

def update(self, initial_value=None, volatility=None, lamb=None,
           mu=None, delta=None, final_date=None):
    if initial_value is not None:
        self.initial_value = initial_value
    if volatility is not None:
        self.volatility = volatility
    if lamb is not None:
        self.lamb = lamb
    if mu is not None:
        self.mu = mu
    if delta is not None:
        self.delt = delta
    if final_date is not None:
        self.final_date = final_date
    self.instrument_values = None

def generate_paths(self, fixed_seed=False, day_count=365.):
    if self.time_grid is None:
        # method from generic simulation class
        self.generate_time_grid()
    # number of dates for time grid
```

```python
        M = len(self.time_grid)
        # number of paths
        I = self.paths
        # ndarray initialization for path simulation
        paths = np.zeros((M, I))
        # initialize first date with initial_value
        paths[0] = self.initial_value
        if self.correlated is False:
            # if not correlated, generate random numbers
            sn1 = sn_random_numbers((1, M, I),
                                    fixed_seed=fixed_seed)
        else:
            # if correlated, use random number object as provided
            # in market environment
            sn1 = self.random_numbers

        # standard normally distributed pseudo-random numbers
        # for the jump component
        sn2 = sn_random_numbers((1, M, I),
                                fixed_seed=fixed_seed)
        rj = self.lamb * (np.exp(self.mu + 0.5 * self.delt ** 2) - 1)

        short_rate = self.discount_curve.short_rate
        for t in range(1, len(self.time_grid)):
            # select the right time slice from the relevant
            # random number set
            if self.correlated is False:
                ran = sn1[t]
            else:
                # only with correlation in portfolio context
                ran = np.dot(self.cholesky_matrix, sn1[:, t, :])
                ran = ran[self.rn_set]
            dt = (self.time_grid[t] - self.time_grid[t - 1]).days / day_count
            # difference between two dates as year fraction
            poi = np.random.poisson(self.lamb * dt, I)
            # Poisson-distributed pseudo-random numbers for jump component
            paths[t] = paths[t - 1] * (
                np.exp((short_rate - rj -
                        0.5 * self.volatility ** 2) * dt +
                        self.volatility * np.sqrt(dt) * ran) +
                (np.exp(self.mu + self.delt * sn2[t]) - 1) * poi)
        self.instrument_values = paths
```

当然，因为我们现在处理的是不同的模型，所以需要 `dx.market_environment` 对象中的一组不同元素。除了用于 `dx.geometric_brownian_motion` 类（见表 18-1）的元素之

外，还增加了表 18-2 中的 3 种元素，即对数正态跳跃成分的参数：lambda、mu 和 delta。

表 18-2 **jump_diffusion** 类的特殊市场环境元素

元素	类型	强制	描述
lambda	常量	是	跳跃密度（概率，按年）
mu	常量	是	预期跳跃规模
delta	常量	是	跳跃规模的标准差

由于跳跃成分的存在，所以为了生成路径，这个类需要更多随机数。generate_paths() 方法的内嵌注释强调了这些额外的随机数生成的两个位置。泊松分布随机数的生成参见第 12 章。

18.4.2 用例

下面的交互式会话说明模拟类 dx.jump_diffusion 的使用方法。我们使用前面为 GBM 对象定义的 dx.market_environment 对象：

```
In [47]: me_jd = market_environment('me_jd', dt.datetime(2020, 1, 1))

In [48]: me_jd.add_constant('lambda', 0.3) ❶
         me_jd.add_constant('mu', -0.75) ❶
         me_jd.add_constant('delta', 0.1) ❶

In [49]: me_jd.add_environment(me_gbm) ❷

In [50]: from jump_diffusion import jump_diffusion

In [51]: jd = jump_diffusion('jd', me_jd)

In [52]: %time paths_3 = jd.get_instrument_values() ❸
         CPU times: user 28.6 ms, sys: 4.37 ms, total: 33 ms
         Wall time: 49.4 ms

In [53]: jd.update(lamb=0.9) ❹

In [54]: %time paths_4 = jd.get_instrument_values() ❺
         CPU times: user 29.7 ms, sys: 3.58 ms, total: 33.3 ms
         Wall time: 66.7 ms
```

❶ dx.jump_diffusion 对象的 3 个附加参数。它们特定于模拟类。

❷ 在现有环境上添加一个完整的环境。

❸ 用基本参数模拟路径。

❹ 增大扩散密度参数。

❺ 以更新的参数模拟路径。

图 18-2 比较分别来自高密度和低密度（跳跃概率）集合的模拟路径。在图 18-2 中可以看到，低密度的情况下跳跃较少，而高密度的情况下有多处跳跃：

```
In [55]: plt.figure(figsize=(10, 6))
         p1 = plt.plot(gbm.time_grid, paths_3[:, :10], 'b')
         p2 = plt.plot(gbm.time_grid, paths_4[:, :10], 'r-.')
         l1 = plt.legend([p1[0], p2[0]],
                         ['low intensity', 'high intensity'], loc=3)
         plt.gca().add_artist(l1)
         plt.xticks(rotation=30);
```

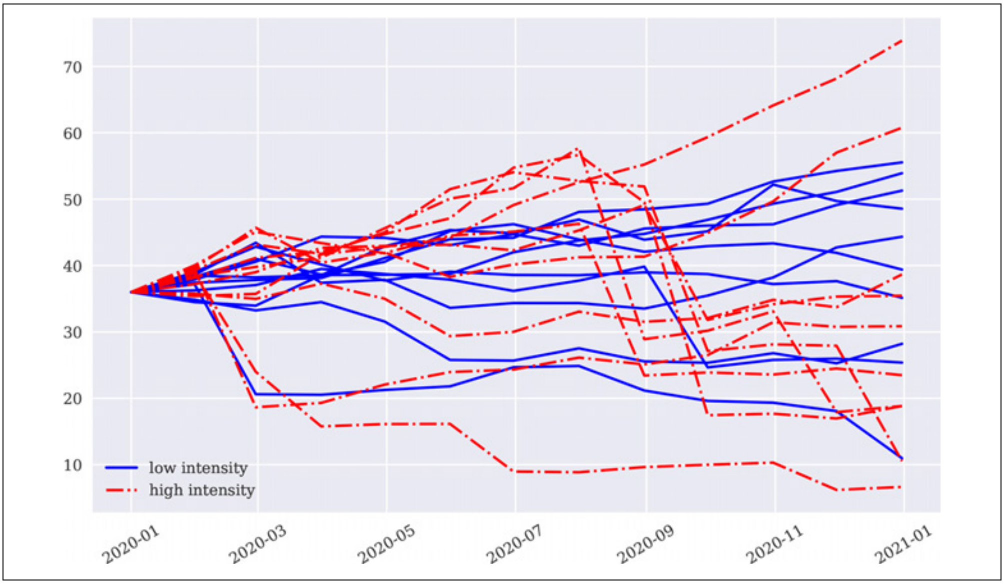


图 18-2　来自跳跃扩散模拟类的模拟路径

18.5　平方根扩散

我们要模拟的第三种随机过程是 Cox、Ingersoll 和 Ross（1985）用于建立随机短期利率模型的平方根扩散。公式 18-5 展示了这种过程的随机微分方程（进一步的细节参见第 12 章的公式 12-4）。

公式 18-5　平方根扩散的随机微分方程

$$dx_t = \kappa(\theta - x_t)dt + \sigma\sqrt{x_t}dZ_t$$

我们的代码使用公式 18-6 提供的离散化格式（参见第 12 章的公式 12-5，以及公式 12-6 中的精确替代格式）。

公式 18-6　平方根扩散的欧拉离散化

$$\tilde{x}_{t_{m+1}} = \tilde{x}_{t_m} + \kappa(\theta - \tilde{x}_s^+)(t_{m+1} - t_m) + \sigma\sqrt{\tilde{x}_s^+}\sqrt{t_{m+1} - t_m}\, z_t$$
$$x_{t_{m+1}} = \tilde{x}^{`}_{t_{m`1}}$$

18.5.1　模拟类

下面是第三个（也是最后一个）模拟类 `dx.square_root_diffusion` 的 Python 代码。当然，除了模型和离散化格式不同之外，和其他两个特殊类相比，这个类没有什么新内容。

```
#
# DX Package
#
# Simulation Class -- Square-Root Diffusion
#
# square_root_diffusion.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class

class square_root_diffusion(simulation_class):
    ''' Class to generate simulated paths based on
    the Cox-Ingersoll-Ross (1985) square-root diffusion model.

    Attributes
    ==========
    name : string
        name of the object
    mar_env : instance of market_environment
        market environment data for simulation
    corr : Boolean
        True if correlated with other model object

    Methods
    =======
    update :
        updates parameters
```

```
    generate_paths :
        returns Monte Carlo paths given the market environment
    '''

    def __init__(self, name, mar_env, corr=False):
        super(square_root_diffusion, self).__init__(name, mar_env, corr)
        # additional parameters needed
        self.kappa = mar_env.get_constant('kappa')
        self.theta = mar_env.get_constant('theta')

    def update(self, initial_value=None, volatility=None, kappa=None,
               theta=None, final_date=None):
        if initial_value is not None:
            self.initial_value = initial_value
        if volatility is not None:
            self.volatility = volatility
        if kappa is not None:
            self.kappa = kappa
        if theta is not None:
            self.theta = theta
        if final_date is not None:
            self.final_date = final_date
        self.instrument_values = None

    def generate_paths(self, fixed_seed=True, day_count=365.):
        if self.time_grid is None:
            self.generate_time_grid()
        M = len(self.time_grid)
        I = self.paths
        paths = np.zeros((M, I))
        paths_ = np.zeros_like(paths)
        paths[0] = self.initial_value
        paths_[0] = self.initial_value
        if self.correlated is False:
            rand = sn_random_numbers((1, M, I),
                                     fixed_seed=fixed_seed)
        else:
            rand = self.random_numbers

        for t in range(1, len(self.time_grid)):
            dt = (self.time_grid[t] - self.time_grid[t - 1]).days / day_count
            if self.correlated is False:
                ran = rand[t]
            else:
                ran = np.dot(self.cholesky_matrix, rand[:, t, :])
```

```
                ran = ran[self.rn_set]

            # full truncation Euler discretization
            paths_[t] = (paths_[t - 1] + self.kappa *
                         (self.theta - np.maximum(0, paths_[t - 1, :])) * dt +
                         np.sqrt(np.maximum(0, paths_[t - 1, :])) *
                         self.volatility * np.sqrt(dt) * ran)
            paths[t] = np.maximum(0, paths_[t])
        self.instrument_values = paths
```

表 18-3 列出了这个类特有的两个市场环境元素。

表 18-3　**dx.square_root_diffusion** 类市场环境的特定元素

元　　素	类　　型	强　　制	描　　述
kappa	常量	是	均值回归因子
theta	常量	是	过程长期均值

18.5.2　用例

一个相当简短的用例就能说明模拟类的使用方法。和往常一样，我们需要一个市场环境，例如用于建立波动率（指数）过程模型的环境：

```
In [56]: me_srd = market_environment('me_srd', dt.datetime(2020, 1, 1)) ❶

In [57]: me_srd.add_constant('initial_value', .25)
         me_srd.add_constant('volatility', 0.05)
         me_srd.add_constant('final_date', dt.datetime(2020, 12, 31))
         me_srd.add_constant('currency', 'EUR')
         me_srd.add_constant('frequency', 'W')
         me_srd.add_constant('paths', 10000)

In [58]: me_srd.add_constant('kappa', 4.0)
         me_srd.add_constant('theta', 0.2)

In [59]: me_srd.add_curve('discount_curve', constant_short_rate('r', 0.0)) ❷

In [60]: from square_root_diffusion import square_root_diffusion

In [61]: srd = square_root_diffusion('srd', me_srd) ❸

In [62]: srd_paths = srd.get_instrument_values()[:, :10] ❹
```

❶ dx.square_root_diffusion 对象的附加参数。

❷ `discount_curve` 对象在默认情况下是必需的，但在这一模拟中不需要。

❸ 实例化对象。

❹ 模拟路径，并选择 10。

图 18-3 显示，单一模拟路径平均回归到长期均值 theta（虚线，假定为 0.2），这说明了均值回归特性：

```
In [55]: plt.figure(figsize=(10, 6))
         p1 = plt.plot(gbm.time_grid, paths_3[:, :10], 'b')
         p2 = plt.plot(gbm.time_grid, paths_4[:, :10], 'r-.')
         l1 = plt.legend([p1[0], p2[0]],
                         ['low intensity', 'high intensity'], loc=3)
         plt.gca().add_artist(l1)
         plt.xticks(rotation=30);
```

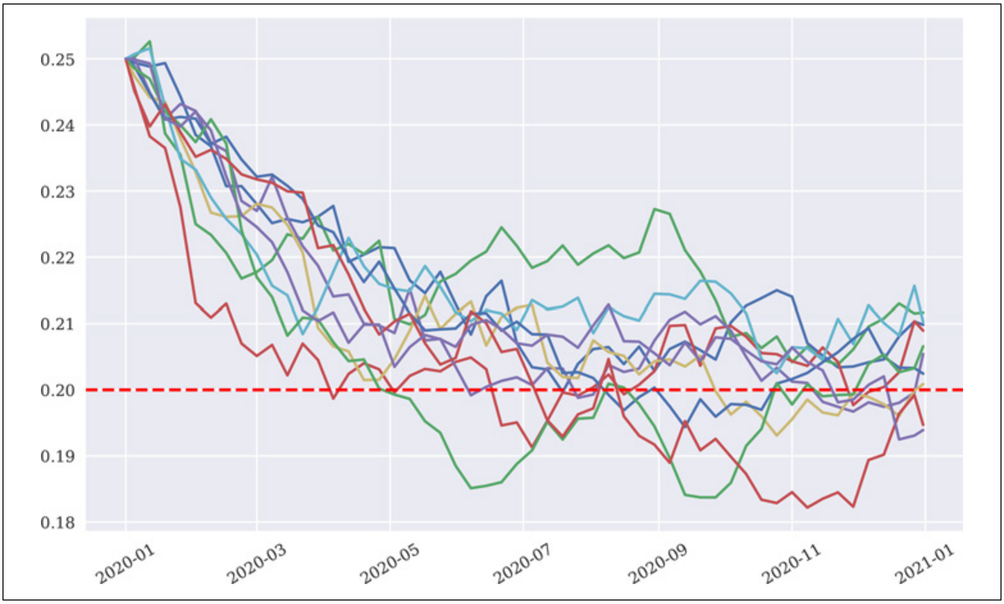


图 18-3　来自平方根扩散模拟类的模拟路径（虚线为长期均值 theta）

18.6　结语

本章开发了我们感兴趣的 3 种随机过程（几何布朗运动、跳跃扩散和平方根扩散）的模拟所需的所有工具和类，还介绍了一个可以方便地生成标准正态分布随机数的函数，然后介绍了一个通用模型模拟类。以此为基础，本章引入了 3 个专用模拟类并提供了这些类的用例。

为了简化未来的导入，我们同样可以使用一个包装器模块 `dx_simulation.py`：

```
#
# DX Package
#
# Simulation Functions & Classes
#
# dx_simulation.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
import pandas as pd
from dx_frame import *
from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class
from geometric_brownian_motion import geometric_brownian_motion
from jump_diffusion import jump_diffusion
from square_root_diffusion import square_root_diffusion
```

和第一个包装器模块 `dx_frame.py` 一样，这样做的好处是用一条 import 语句就可以一次导入所有模拟组件：

```
from dx_simulation import *
```

由于 `dx_simulation.py` 还导入来自 `dx_frame.py` 的所有组件，这样的导入实际上输出了目前为止开发的所有功能。对于 dx 目录中的增强版 `__init__.py` 文件也是如此：

```
#
# DX Package
# packaging file
# __init__.py
#
import numpy as np
import pandas as pd
import datetime as dt

# frame
from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment

# simulation
from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class
```

```
from geometric_brownian_motion import geometric_brownian_motion
from jump_diffusion import jump_diffusion
from square_root_diffusion import square_root_diffusion
```

18.7　延伸阅读

下面是本章介绍的主题的一些实用参考书：

- Glasserman, Paul (2004). *Monte Carlo Methods in Financial Engineering*. New York: Springer.
- Hilpisch, Yves (2015): *Derivatives Analytics with Python*. Chichester, England: Wiley Finance.

本章引用的原创论文如下：

- Black, Fischer, and Myron Scholes (1973). “The Pricing of Options and Corporate Liabilities.” *Journal of Political Economy*, Vol. 81, No. 3, pp. 638-659.
- Cox, John, Jonathan Ingersoll, and Stephen Ross (1985). “A Theory of the Term Structure of Interest Rates.” *Econometrica*, Vol. 53, No. 2, pp. 385-407.
- Merton, Robert (1973). “Theory of Rational Option Pricing.” *Bell Journal of Economics and Management Science*, Vol. 4, pp. 141-183.
- Merton, Robert (1976). “Option Pricing When the Underlying Stock Returns Are Discontinuous.” *Journal of Financial Economics*, Vol. 3, No. 3, pp. 125-144.

第 19 章

衍生品估值

衍生品是一个巨大、复杂的问题。

——贾德·格雷格

期权和衍生品估值很久以来都属于华尔街“火箭科学家”（拥有物理学或者与数学相关的类似高要求学科博士学位的人们）的领域。然而，通过蒙特卡洛模拟等数值方法，这些模型的应用通常不像理论模型本身那么复杂。

对于采用欧式行权方法的期权和衍生品（即只可能在某个预先确定的日期行权）来说，上述说法尤为正确。对于采用美式行权方法的期权和衍生品来说则稍微复杂一些，这种行权可以在预先规定的一段时间内进行。本章介绍和使用最小二乘蒙特卡洛（LSM）算法，这种算法已经成为基于蒙特卡洛模拟的美式期权估值中的基准算法。

本章在结构上类似于第 18 章，首先介绍一个通用估值类，然后提供两个专用估值类，一个用于欧式行权，另一个用于美式行权。通用估值类包含以数值形式估算重要期权“希腊字母”（Delta 和 Vega）的方法。因此，估值类不仅对估值很重要，对风险管理也很重要。

本章的结构如下。

通用估值类

　　本节介绍通用估值类，其他特殊类将从该类中继承。

欧式行权

　　本节介绍的是欧式行权方式的期权和衍生品估值类。

美式行权

　　本节介绍的是美式行权方式的期权和衍生品估值类。

19.1　通用估值类

与通用模拟类一样，我们仅提供少数几个输入（本例是 4 个）来实例化一个估值类对象。

name

　　表示模型模拟对象名称的字符串对象。

underlying

　　代表标的的模拟类实例。

mar_env

　　dx.market_environment 类实例。

payoff_func

　　包含期权/衍生品收益函数的 Python 字符串。

通用类有以下 3 个方法。

update()

　　这个方法更新选择的估值参数（属性）。

delta()

　　这个方法计算期权/衍生品的 Delta 值。

vega()

　　这个方法计算期权/衍生品的 Vega 值。

有了前面几章中学习到的 DX 库背景知识，下面介绍的通用估值类几乎不言自明；在合适的地方还提供了内嵌注释。我们仍然首先提供完整的类，然后在后续的内容中重点介绍选择的主题。

```
#
# DX Package
#
# Valuation -- Base Class
#
# valuation_class.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
```

```python
class valuation_class(object):
    ''' Basic class for single-factor valuation.

    Attributes
    ==========
    name: str
        name of the object
    underlying: instance of simulation class
        object modeling the single risk factor
    mar_env: instance of market_environment
        market environment data for valuation
    payoff_func: str
        derivatives payoff in Python syntax
        Example: 'np.maximum(maturity_value - 100, 0)'
        where maturity_value is the NumPy vector with
        respective values of the underlying
        Example: 'np.maximum(instrument_values - 100, 0)'
        where instrument_values is the NumPy matrix with
        values of the underlying over the whole time/path grid

    Methods
    =======
    update:
        updates selected valuation parameters
    delta:
        returns the delta of the derivative
    vega:
        returns the vega of the derivative
    '''

    def __init__(self, name, underlying, mar_env, payoff_func=''):
        self.name = name
        self.pricing_date = mar_env.pricing_date
        try:
            # strike is optional
            self.strike = mar_env.get_constant('strike')
        except:
            pass
        self.maturity = mar_env.get_constant('maturity')
        self.currency = mar_env.get_constant('currency')
        # simulation parameters and discount curve from simulation object
        self.frequency = underlying.frequency
        self.paths = underlying.paths
        self.discount_curve = underlying.discount_curve
        self.payoff_func = payoff_func
```

```
        self.underlying = underlying
        # provide pricing_date and maturity to underlying
        self.underlying.special_dates.extend([self.pricing_date,
                                              self.maturity])

    def update(self, initial_value=None, volatility=None,
               strike=None, maturity=None):
        if initial_value is not None:
            self.underlying.update(initial_value=initial_value)
        if volatility is not None:
            self.underlying.update(volatility=volatility)
        if strike is not None:
            self.strike = strike
        if maturity is not None:
            self.maturity = maturity
            # add new maturity date if not in time_grid
            if maturity not in self.underlying.time_grid:
                self.underlying.special_dates.append(maturity)
                self.underlying.instrument_values = None

    def delta(self, interval=None, accuracy=4):
        if interval is None:
            interval = self.underlying.initial_value / 50.
        # forward-difference approximation
        # calculate left value for numerical delta
        value_left = self.present_value(fixed_seed=True)
        # numerical underlying value for right value
        initial_del = self.underlying.initial_value + interval
        self.underlying.update(initial_value=initial_del)
        # calculate right value for numerical delta
        value_right = self.present_value(fixed_seed=True)
        # reset the initial_value of the simulation object
        self.underlying.update(initial_value=initial_del - interval)
        delta = (value_right - value_left) / interval
        # correct for potential numerical errors
        if delta < -1.0:
            return -1.0
        elif delta > 1.0:
            return 1.0
        else:
            return round(delta, accuracy)

    def vega(self, interval=0.01, accuracy=4):
        if interval < self.underlying.volatility / 50.:
            interval = self.underlying.volatility / 50.
```

```
        # forward-difference approximation
        # calculate the left value for numerical vega
        value_left = self.present_value(fixed_seed=True)
        # numerical volatility value for right value
        vola_del = self.underlying.volatility + interval
        # update the simulation object
        self.underlying.update(volatility=vola_del)
        # calculate the right value for numerical vega
        value_right = self.present_value(fixed_seed=True)
        # reset volatility value of simulation object
        self.underlying.update(volatility=vola_del - interval)
        vega = (value_right - value_left) / interval
        return round(vega, accuracy)
```

通用类 `dx.valuation_class` 包含的主题之一是“希腊字母”的估算。这是我们应该仔细观察的。为此，考虑一个代表期权现值的连续可微分函数 $V(S_0, \sigma_0)$。期权的 Delta 值定义为标的当前价值 S_0 的一阶偏微分，即 $\Delta = \frac{\partial V(\cdot)}{\partial S_0}$。

假定现在我们从蒙特卡洛估值（参见第 12 章和本章后续的内容）中得到一个期权价值的蒙特卡洛估算函数 $\bar{V}(S_0, \sigma_0)$。期权 Delta 值的数值逼近由公式 19-1 给出[1]。这是通用估值类 `delta()`方法所实现的功能。该方法假定存在 `present_value()`方法，`present_value()`方法返回给定某组参数值下的蒙特卡洛估算函数。

公式 19-1　期权 Delta 值的数值解

$$\bar{\Delta} = \frac{\bar{V}(S_0 + \Delta S, \sigma_0) - \bar{V}(S_0, \sigma_0)}{\Delta S}, \Delta S > 0$$

类似地，金融工具的 Vega 值定义为在当前（即期）波动率 σ_0 下现值的一阶偏微分，即 $\mathbf{V} = \frac{\partial V(\cdot)}{\partial \sigma_0}$。同样假设存在期权价值的蒙特卡洛估算函数，公式 19-2 提供了 Vega 的数值逼近公式。这是 `dx.valuation_class` 类的 `vega()`方法实现的功能。

公式 19-2　期权 Vega 值的数值解

$$\mathbf{V} = \frac{\bar{V}(S_0, \sigma_0 + \Delta\sigma) - \bar{V}(S_0, \sigma_0)}{\Delta\sigma}, \Delta\sigma > 0$$

注意，Delta 和 Vega 的讨论只基于有一个可微分函数或者期权现值的蒙特卡洛估算函数存在的情况。这是我们可以定义方法，以数值化方式估算这些数量，而无须了解蒙

1　通过蒙特卡洛模拟估算“希腊字母”的细节参见 Glassman(2004)的第 7 章。我们在此只使用前向差分格式，因为这只会导致一次额外的模拟和期权重新估值。例如，中心差分逼近法将导致两次期权重估值，从而造成更大的计算负担。——原注

特卡洛估算函数精确定义和数值实现的原因。

19.2 欧式行权

我们要专门化通用估值类的第一种情况是欧式行权。为此，考虑生成期权价值蒙特卡洛估算函数的简化方法，如下所列。

- 进行 I 次风险中立测度下对应标的风险因子 S 的模拟，得出同样多的期权 T 到期日标的物模拟价值——即 $\bar{S}_T(i), i \in \{1, 2, \cdots, I\}$ 。
- 计算到期日每种模拟标的物价值的期权收益 h_T——$h_T(\bar{S}_T(i)), i \in \{1, 2, \cdots, I\}$ 。
- 得出期权现值的蒙特卡洛估算函数 $\bar{V}_0 \equiv e^{-rT} \frac{1}{I} \sum_{i=1}^{I} h_T(\bar{S}_T(i))$ 。

19.2.1 估值类

下面的代码展示了根据上述方法实现 present_value()方法的类。此外，该类还包含 generate_payoff()方法以生成模拟路径和给定路径的期权收益。这是蒙特卡洛估算函数的基础。

```
#
# DX Package
#
# Valuation -- European Exercise Class
#
# valuation_mcs_european.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from valuation_class import valuation_class

class valuation_mcs_european(valuation_class):
    ''' Class to value European options with arbitrary payoff
    by single-factor Monte Carlo simulation.

    Methods
    =======
    generate_payoff:
        returns payoffs given the paths and the payoff function
    present_value:
```

```
        returns present value (Monte Carlo estimator)
    '''

    def generate_payoff(self, fixed_seed=False):
        '''
        Parameters
        ==========
        fixed_seed: bool
            use same/fixed seed for valuation
        '''
        try:
            # strike is optional
            strike = self.strike
        except:
            pass
        paths = self.underlying.get_instrument_values(fixed_seed=fixed_seed)
        time_grid = self.underlying.time_grid
        try:
            time_index = np.where(time_grid == self.maturity)[0]
            time_index = int(time_index)
        except:
            print('Maturity date not in time grid of underlying.')
        maturity_value = paths[time_index]
        # average value over whole path
        mean_value = np.mean(paths[:time_index], axis=1)
        # maximum value over whole path
        max_value = np.amax(paths[:time_index], axis=1)[-1]
        # minimum value over whole path
        min_value = np.amin(paths[:time_index], axis=1)[-1]
        try:
            payoff = eval(self.payoff_func)
            return payoff
        except:
            print('Error evaluating payoff function.')

    def present_value(self, accuracy=6, fixed_seed=False, full=False):
        '''
        Parameters
        ==========
        accuracy: int
            number of decimals in returned result
        fixed_seed: bool
            use same/fixed seed for valuation
        full: bool
            return also full 1d array of present values
```

```
        '''
        cash_flow = self.generate_payoff(fixed_seed=fixed_seed)
        discount_factor = self.discount_curve.get_discount_factors(
            (self.pricing_date, self.maturity))[0, 1]
        result = discount_factor * np.sum(cash_flow) / len(cash_flow)
        if full:
            return round(result, accuracy), discount_factor * cash_flow
        else:
            return round(result, accuracy)
```

`generate_payoff()`方法提供了一些特殊的对象用于期权收益的定义。

- `strike` 是期权的行权价。
- `maturity_value` 表示期权到期日标的物模拟价值。
- `mean_value` 是从今天到到期日整条路径上标的物的平均价值。
- `max_value` 是整条路径上标的物的最大价值。
- `min_value` 是整条路径上标的物的最小价值。

后 3 个对象特别考虑了具有亚洲式（回顾式）特征的期权的高效处理。

> **灵活收益**
> 欧式行权的期权及衍生品估值方法相当灵活，可以定义任意的收益函数。这允许我们建立有条件行权（例如期权）和无条件行权（例如远期合约）的模型，也可以加入其他风格的收益因素，例如回顾式特征。

19.2.2 用例

估值类 `dx.valuation_mcs_european` 的应用最好通过一个特定的用例来说明。但是，在实例化估值之前，我们需要一个模拟对象——用于估算期权价值的标的。从第 18 章起，我们就使用 `dx.gemetric_brownian_motion` 类建立标的物模型。我们还使用对应用例的示例参数：

```
In [64]: me_gbm = market_environment('me_gbm', dt.datetime(2020, 1, 1))

In [65]: me_gbm.add_constant('initial_value', 36.)
         me_gbm.add_constant('volatility', 0.2)
         me_gbm.add_constant('final_date', dt.datetime(2020, 12, 31))
         me_gbm.add_constant('currency', 'EUR')
         me_gbm.add_constant('frequency', 'M')
         me_gbm.add_constant('paths', 10000)
```

```
In [66]: csr = constant_short_rate('csr', 0.06)

In [67]: me_gbm.add_curve('discount_curve', csr)

In [68]: gbm = geometric_brownian_motion('gbm', me_gbm)
```

除了模拟对象之外，我们需要为期权提供一个市场环境，其中必须包含至少一个 `maturity` 和一个 `currency`。我们还可以选择提供 `strike` 参数：

```
In [69]: me_call = market_environment('me_call', me_gbm.pricing_date)

In [70]: me_call.add_constant('strike', 40.)
         me_call.add_constant('maturity', dt.datetime(2020, 12, 31))
         me_call.add_constant('currency', 'EUR')
```

核心元素当然是收益函数，在这里以包含 Python 代码的一个字符串对象形式提供，这些代码由 `eval()` 函数进行求值。我们打算定义欧式看涨期权，这种期权的收益为 $h_T = \max(S_T - K, 0)$，S_T是到期日标的价值，K 是期权行权价。在 Python 和 NumPy 中，所有模拟值都采用向量化存储，其形式如下：

```
In [71]: payoff_func = 'np.maximum(maturity_value - strike, 0)'
```

现在，将所有成分组合起来，就可以从 `dx.valuation_mcs_european` 类实例化一个对象。有了这个估值对象后，所有感兴趣的数量都只需要一次方法调用即可得到：

```
In [72]: from valuation_mcs_european import valuation_mcs_european

In [73]: eur_call = valuation_mcs_european('eur_call', underlying=gbm,
                                mar_env=me_call, payoff_func=payoff_func)

In [74]: %time eur_call.present_value() ❶
         CPU times: user 14.8 ms, sys: 4.06 ms, total: 18.9 ms
         Wall time: 43.5 ms

Out[74]: 2.146828

In [75]: %time eur_call.delta() ❷
         CPU times: user 12.4 ms, sys: 2.68 ms, total: 15.1 ms
         Wall time: 40.1 ms

Out[75]: 0.5155

In [76]: %time eur_call.vega() ❸
         CPU times: user 21 ms, sys: 2.72 ms, total: 23.7 ms
         Wall time: 89.9 ms

Out[76]: 14.301
```

❶ 估算欧式看涨期权的现值。

❷ 估算期权的 Delta 值，看涨期权的 Delta 值为正。

❸ 估算期权的 Vega 值，看涨与看跌期权的 Vega 值都为正。

一旦拥有了估值对象，就很容易实施现值和“希腊字母”的更全面分析。下面的代码计算标的初值为 34～46 欧元时的现值、Delta 和 Vega。结果如图 19-1 所示。

```
In [77]: %%time
         s_list = np.arange(34., 46.1, 2.)
         p_list = []; d_list = []; v_list = []
         for s in s_list:
             eur_call.update(initial_value=s)
             p_list.append(eur_call.present_value(fixed_seed=True))
             d_list.append(eur_call.delta())
             v_list.append(eur_call.vega())
         CPU times: user 374 ms, sys: 8.82 ms, total: 383 ms
         Wall time: 609 ms

In [78]: from plot_option_stats import plot_option_stats

In [79]: plot_option_stats(s_list, p_list, d_list, v_list)
```

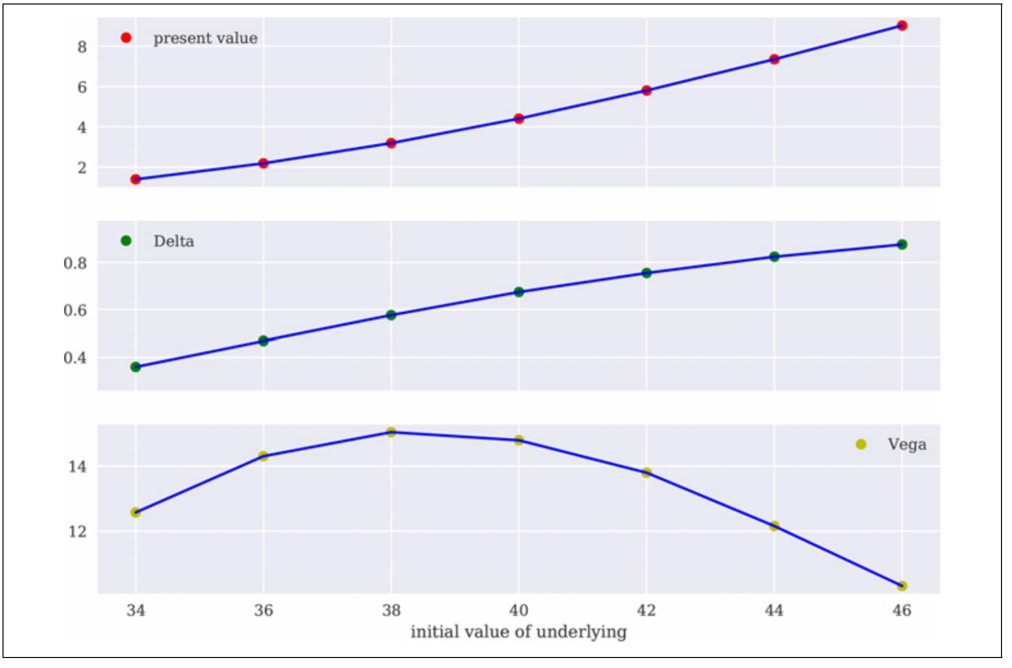


图 19-1　欧式看涨期权的现值、Delta 和 Vega 的估算

可使用助手函数 plot_option_stats()进行可视化：

```
#
# DX Package
#
# Valuation -- Plotting Options Statistics
#
# plot_option_stats.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import matplotlib.pyplot as plt

def plot_option_stats(s_list, p_list, d_list, v_list):
    ''' Plots option prices, deltas, and vegas for a set of
    different initial values of the underlying.

    Parameters
    ==========
    s_list: array or list
        set of initial values of the underlying
    p_list: array or list
        present values
    d_list: array or list
        results for deltas
    v_list: array or list
        results for vegas
    '''
    plt.figure(figsize=(10, 7))
    sub1 = plt.subplot(311)
    plt.plot(s_list, p_list, 'ro', label='present value')
    plt.plot(s_list, p_list, 'b')
    plt.legend(loc=0)
    plt.setp(sub1.get_xticklabels(), visible=False)
    sub2 = plt.subplot(312)
    plt.plot(s_list, d_list, 'go', label='Delta')
    plt.plot(s_list, d_list, 'b')
    plt.legend(loc=0)
    plt.ylim(min(d_list) - 0.1, max(d_list) + 0.1)
    plt.setp(sub2.get_xticklabels(), visible=False)
    sub3 = plt.subplot(313)
    plt.plot(s_list, v_list, 'yo', label='Vega')
    plt.plot(s_list, v_list, 'b')
    plt.xlabel('initial value of underlying')
    plt.legend(loc=0)
```

这说明，DX 库的使用尽管涉及很多数值计算，但是归结起来，采用的方法可以与拥有封闭式期权定价公式的情况媲美。这种方法不仅适用于目前考虑的简单收益。而且可以处理复杂得多的收益。

为此，我们考虑常规和亚式收益的组合。处理方式与分析方式相同，并且都基本独立于定义的收益类型。图 19-2 显示，标的初始价值接近行权价（40）时 Delta 变为 1。从这个特定点起，标的初始价值的每次（鞅）增加均会导致期权价值有同等的（鞅）增加：

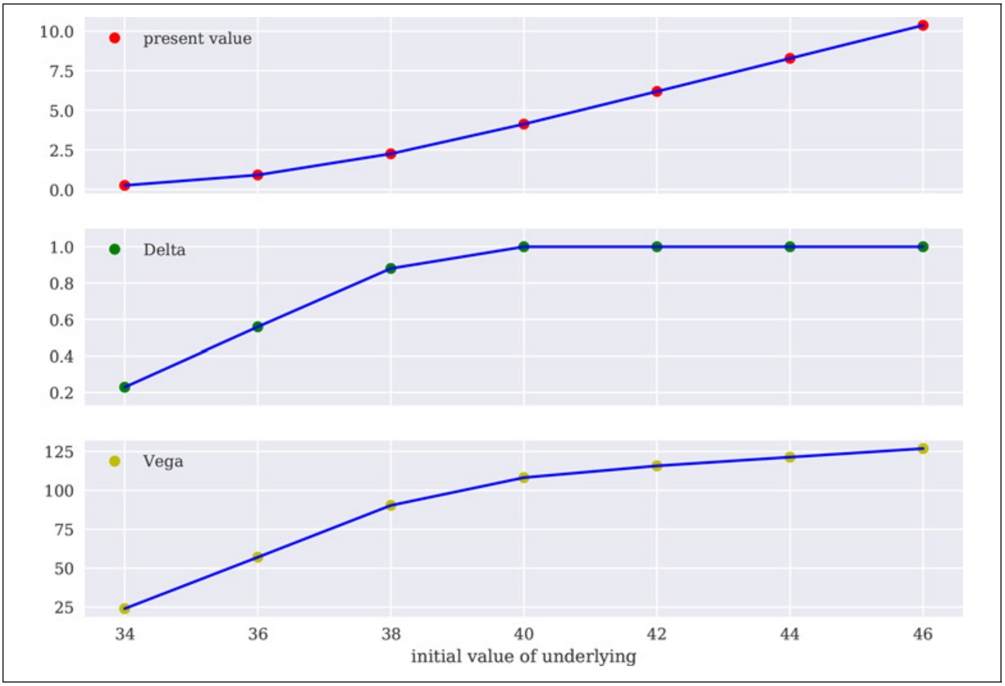


图 19-2　欧式-亚式看涨期权的现值、Delta 和 Vega 估算

```
In [80]: payoff_func = 'np.maximum(0.33 * '
         payoff_func += '(maturity_value + max_value) - 40, 0)' ❶

In [81]: eur_as_call = valuation_mcs_european('eur_as_call', underlying=gbm,
                                 mar_env=me_call, payoff_func=payoff_func)

In [82]: %%time
         s_list = np.arange(34., 46.1, 2.)
         p_list = []; d_list = []; v_list = []
         for s in s_list:
             eur_as_call.update(s)
             p_list.append(eur_as_call.present_value(fixed_seed=True))
             d_list.append(eur_as_call.delta())
             v_list.append(eur_as_call.vega())
```

```
CPU times: user 319 ms, sys: 14.2 ms, total: 333 ms
Wall time: 488 ms

In [83]: plot_option_stats(s_list, p_list, d_list, v_list)
```

❶ 收益取决于模拟到期日价值和模拟路径上的最大值。

19.3 美式行权

美式期权（或者百慕大式期权[1]）的估值比欧式期权要复杂得多。因此，在介绍估值类之前，我们必须首先介绍更多估值理论。

19.3.1 最小二乘蒙特卡洛方法

Cox、Ross 和 Rubinstein（1979）提出了他们的二项式模型，在相同框架里用简单的数值化方法估算欧式和美式期权的价值，但是只有 Longstaff-Schwartz（2001）的模型完美地解决了用蒙特卡洛模拟（MCS）估计美式期权价值的问题。主要的问题是，MCS 本身是前移式算法，而美式期权的估值通常利用倒推法实现——从到期日开始估算美式期权的持续价值并倒推出现值。

Longstaff-Schwartz（2001）模型的深意在于使用了普通最小二乘回归[2]，根据所有可用模拟值的交点估算持续价值。对每条路径考虑如下因素：

- 标的模拟价值；
- 期权内在价值；
- 特定路径的实际持续价值。

在离散时间中，百慕大期权（以及极限情况下的美式期权）的价值由最优截止问题给出。对于有限时点集合 $0< t_1<t_2<\cdots<T$，[3]最优截止问题如公式 19-3 所示。

公式 19-3　百慕大期权离散时间中的最优截止问题

$$V_0 = \sup_{\tau\in\{0,t_1,t_2,\cdots,T\}} \mathrm{e}^{-r\tau} E_0^Q(h_t(S_\tau))$$

公式 19-4 提供了日期 $0\leqslant t_m< T$ 时美式期权的持续价值。这就是在美式期权 V_{tm+1} 后续日期价值的鞅测度下，日期 t_m 时的风险中立预期。

1　美式行权指的是行权可以在一段固定期间中的每个时间点（至少在交易时间内）中进行的情况。百慕大行权通常指的是有多个离散行权日的情况，在数值应用中，美式行权由百慕大行权近似计算，在极限的情况下，行权日期的数量可能不限。——原注

2　这就是这种算法通常缩写为 LSM（最小二乘蒙特卡洛）的原因。——原注

3　Kohler（2010）提供了美式期权估值理论的简洁概述，尤其是基于回归方法的使用。——原注

公式 19-4　美式期权持续价值

$$C_{t_m}(s)=\mathrm{e}^{-r(t_{m+1}-t_m)}E_{t_m}^{Q}(V_{t_{m`1}}(S_{t_{m`1}})\,|\,S_{t_m}=s)$$

美式期权 V_{tm} 在日期 t_m 时的价值也可以用公式 19-5 中的公式表示——立即行权的收益（内在价值）和不行权预期收益（持续价值）两者的最大值。

公式 19-5　美式权证在任何给定日期的价值

$$V_{t_m}=\max(h_{t_m}(s),C_{t_m}(s))$$

在公式 19-5 中，内在价值很容易计算。持续价值则稍微有些难度，Longstaff- Schwartz（2001）模型通过公式 19-6 提供的回归公式来近似计算该值。其中，i 表示当前模拟路径，D 是所使用回归基函数的数量，α^*是最优回归参数，b_d 是编号为 d 的回归函数。

公式 19-6　基于回归的持续价值近似计算

$$\bar{C}_{t_m,i}=\sum_{d=1}^{D}\alpha^*_{d,t_m}\bullet b_d(S_{t_m,i})$$

最优回归参数是公式 19-7 提出的最小二乘回归问题的解。式中，$Y_{t_m,i}\equiv \mathrm{e}^{-r(t_{m+1}-t_m)}V_{t_{m+1},i}$ 是路径 i 在日期 t_m 时的实际持续价值（不是回归/估算的价值）。

公式 19-7　普通最小二乘回归

$$\min_{\alpha_{1,t_m},\cdots,\alpha_{D,t_m}}\frac{1}{I}\sum_{i=1}^{I}\left(Y_{t_m,i}-\sum_{d=1}^{D}\alpha_{d,t_m}\bullet b_d(S_{t_m,i})\right)^2$$

这就完成了通过 MCS 估算美式期权价值的基本（数学）工具集。

19.3.2　估值类

例 19-4 提供了采用美式行权方式的期权和衍生品的估值类。在 `present_value()`方法中，LSM 算法的实现有一个值得注意的步骤（在内嵌注释中也体现了）：最优决策步骤。这里重要的是，根据做出的决策，LSM 算法采用内在价值或者实际持续价值——而不是估算的持续价值[1]。

```
#
# DX Package
#
# Valuation -- American Exercise Class
#
# valuation_mcs_american.py
#
```

1　参加 Hilpisch(2015)的第 6 章。——原注

```
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from valuation_class import valuation_class


class valuation_mcs_american(valuation_class):
    ''' Class to value American options with arbitrary payoff
    by single-factor Monte Carlo simulation.

    Methods
    =======
    generate_payoff:
        returns payoffs given the paths and the payoff function
    present_value:
        returns present value (LSM Monte Carlo estimator)
        according to Longstaff-Schwartz (2001)
    '''

def generate_payoff(self, fixed_seed=False):
    '''
    Parameters
    ==========
    fixed_seed:
        use same/fixed seed for valuation
    '''
    try:
        # strike is optional
        strike = self.strike
    except:
        pass
    paths = self.underlying.get_instrument_values(fixed_seed=fixed_seed)
    time_grid = self.underlying.time_grid
    time_index_start = int(np.where(time_grid == self.pricing_date)[0])
    time_index_end = int(np.where(time_grid == self.maturity)[0])
    instrument_values = paths[time_index_start:time_index_end + 1]
    payoff = eval(self.payoff_func)
    return instrument_values, payoff, time_index_start, time_index_end

def present_value(self, accuracy=6, fixed_seed=False, bf=5, full=False):
    '''
    Parameters
    ==========
    accuracy: int
```

```
        number of decimals in returned result
    fixed_seed: bool
        use same/fixed seed for valuation
    bf: int
        number of basis functions for regression
    full: bool
        return also full 1d array of present values
    '''
    instrument_values, inner_values, time_index_start, time_index_end = \
        self.generate_payoff(fixed_seed=fixed_seed)
    time_list = self.underlying.time_grid[
        time_index_start:time_index_end + 1]
    discount_factors = self.discount_curve.get_discount_factors(
        time_list, dtobjects=True)
    V = inner_values[-1]
    for t in range(len(time_list) - 2, 0, -1):
        # derive relevant discount factor for given time interval
        df = discount_factors[t, 1] / discount_factors[t + 1, 1]
        # regression step
        rg = np.polyfit(instrument_values[t], V * df, bf)
        # calculation of continuation values per path
        C = np.polyval(rg, instrument_values[t])
        # optimal decision step:
        # if condition is satisfied (inner value > regressed cont. value)
        # then take inner value; take actual cont. value otherwise
        V = np.where(inner_values[t] > C, inner_values[t], V * df)
    df = discount_factors[0, 1] / discount_factors[1, 1]
    result = df * np.sum(V) / len(V)
    if full:
        return round(result, accuracy), df * V
    else:
        return round(result, accuracy)
```

19.3.3　用例

正如前面选择的方法，用例可以说明 `dx.valuation_mcs_american` 类的用法。这个用例重复 Longstaff 和 Schwartz（2001）原创论文中表 1 所介绍的所有美式期权价值。标的和前面一样，是一个 `dx.geometric_brownian_motion` 对象。标的物的起始参数如下：

```
In [84]: me_gbm = market_environment('me_gbm', dt.datetime(2020, 1, 1))

In [85]: me_gbm.add_constant('initial_value', 36.)
         me_gbm.add_constant('volatility', 0.2)
         me_gbm.add_constant('final_date', dt.datetime(2021, 12, 31))
```

```
        me_gbm.add_constant('currency', 'EUR')
        me_gbm.add_constant('frequency', 'W')
        me_gbm.add_constant('paths', 50000)

In [86]: csr = constant_short_rate('csr', 0.06)

In [87]: me_gbm.add_curve('discount_curve', csr)

In [88]: gbm = geometric_brownian_motion('gbm', me_gbm)

In [89]: payoff_func = 'np.maximum(strike - instrument_values, 0)'

In [90]: me_am_put = market_environment('me_am_put', dt.datetime(2020, 1, 1))

In [91]: me_am_put.add_constant('maturity', dt.datetime(2020, 12, 31))
         me_am_put.add_constant('strike', 40.)
         me_am_put.add_constant('currency', 'EUR')
```

下一步是根据数值化假设实例化估值对象。美式看跌期权估值花费的时间比欧式期权长得多。我们不仅增加了路径的数量和估值的频率，由于向后推导和每个推导步骤的回归，该算法的计算要求也更高。用数值化方法得到的价值基金与原创论文中报告的准确值 4.478 相当接近：

```
In [92]: from valuation_mcs_american import valuation_mcs_american

In [93]: am_put = valuation_mcs_american('am_put', underlying=gbm,
                             mar_env=me_am_put, payoff_func=payoff_func)

In [94]: %time am_put.present_value(fixed_seed=True, bf=5)
         CPU times: user 1.57 s, sys: 219 ms, total: 1.79 s
         Wall time: 2.01 s

Out[94]: 4.472834
```

由于 LSM 蒙特卡洛估算函数的最初构造，所以它可以表示数学上正确的美式期权价值下界[1]。因此，可以预期，这种数值化估算的结果低于任何数值化实例的真值。作为替代的对偶估算函数还能提供上界[2]。将两者结合起来，两个不同的估算函数可以定义美式期权真值的区间。

这个用例的主要目标是复制原创论文中表 1 的所有美式期权价值。为此，我们只需要结合估算对象和嵌套循环。在最内层循环中，估值对象必须根据当时的参数更新：

1 主要原因是基于持续价值回归估算的“最优行权策略”未能达到“最优”的标准。——原注

2 得出上界的对偶算法和 Python 实现请参见 Hilpisch(2015)的第 6 章。——原注

```
In [95]: %%time
         ls_table = []
         for initial_value in (36., 38., 40., 42., 44.):
             for volatility in (0.2, 0.4):
                 for maturity in (dt.datetime(2020, 12, 31),
                                  dt.datetime(2021, 12, 31)):
                     am_put.update(initial_value=initial_value,
                                   volatility=volatility,
                                   maturity=maturity)
                     ls_table.append([initial_value,
                                       volatility,
                                       maturity,
                                       am_put.present_value(bf=5)])
         CPU times: user 41.1 s, sys: 2.46 s, total: 43.5 s
         Wall time: 1min 30s

In [96]: print('S0 | Vola | T | Value')
         print(22 * '-')
         for r in ls_table:
             print('%d | %3.1f | %d | %5.3f' %
                   (r[0], r[1], r[2].year - 2019, r[3]))
         S0 | Vola | T | Value
         ----------------------
         36 | 0.2  | 1 | 4.447
         36 | 0.2  | 2 | 4.773
         36 | 0.4  | 1 | 7.006
         36 | 0.4  | 2 | 8.377
         38 | 0.2  | 1 | 3.213
         38 | 0.2  | 2 | 3.645
         38 | 0.4  | 1 | 6.069
         38 | 0.4  | 2 | 7.539
         40 | 0.2  | 1 | 2.269
         40 | 0.2  | 2 | 2.781
         40 | 0.4  | 1 | 5.211
         40 | 0.4  | 2 | 6.756
         42 | 0.2  | 1 | 1.556
         42 | 0.2  | 2 | 2.102
         42 | 0.4  | 1 | 4.466
         42 | 0.4  | 2 | 6.049
         44 | 0.2  | 1 | 1.059
         44 | 0.2  | 2 | 1.617
         44 | 0.4  | 1 | 3.852
         44 | 0.4  | 2 | 5.490
```

这些结果是 Longstaff 和 Schwartz（2001）论文中表 1 的简化版本。总体来说，我们的

数值化结果相当接近论文中报告的结果，论文使用了一些不同的参数（例如，他们使用两倍数量的路径）。

在这个用例结束的时候要注意，美式期权的“希腊字母”估算在形式上和欧式期权相同——这是我们的方法与其他数值化方法（如二项式模型）相比的重大优势之一：

```
In [97]: am_put.update(initial_value=36.)
         am_put.delta()
Out[97]: -0.4631

In [98]: am_put.vega()
Out[98]: 18.0961
```

最小二乘蒙特卡洛方法

对于美式或百慕大式期权甚至复杂衍生品的估值来说，Longstaff 和 Schwartz（2001）的 LSM 估值算法在数值上很高效。OLS 回归步骤可以根据一种高效的数字方法来实现最优行权策略的近似计算。由于 OLS 回归可以轻松地处理高维数据，所以它成为灵活的衍生品定价方法。

19.4　结语

本章介绍了基于蒙特卡洛模拟的欧式和美式期权数值化估值。本章介绍了一个通用估值类 `dx.valuation_class`，这个类提供了许多方法，例如估算两类期权非常重要的“希腊字母”（delta、vega）的方法，这项方法独立于估值使用的模拟对象（风险因素/随机过程）。

在通用估值类的基础上，本章介绍了两个专用类：`dx.valuation_mcs_european` 和 `dx.valuation_ mcs_american`。欧式期权的估值类主要是第 17 章中介绍的风险中立估值方法简单实现与一个预期项数值化估算（正如第 11 章中所讨论的，是蒙特卡洛模拟的积分）的结合。

美式期权的估值类需要某种基于回归的估值算法（称为最小二乘蒙特卡洛方法 LSM）。这是因为美式期权的估值需要最优行权策略。这在理论上和数值化方法上都更复杂一些，但是，对应类的 `present_value()`方法仍然很简洁。

已经证明，DX 衍生品分析库所采用的方法是很有益的。不需要花费太多的精力，我们就可以估算具有如下特征的一大类期权的价值：

- 单一风险因素期权；
- 欧式或者美式行权；

- 任意收益。

此外，我们可以估算这类期权中非常重要的“希腊字母”。为了简化未来的导入，我们再次使用一个包装器模块 dx_valuation.py：

```
#
# DX Package
#
# Valuation Classes
#
# dx_valuation.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
import pandas as pd

from dx_simulation import *
from valuation_class import valuation_class
from valuation_mcs_european import valuation_mcs_european
from valuation_mcs_american import valuation_mcs_american
```

dx 文件夹中的__init__.py 文件被相应更新：

```
#
# DX Package
# packaging file
# __init__.py
#
import numpy as np
import pandas as pd
import datetime as dt

# frame
from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment
from plot_option_stats import plot_option_stats

# simulation
from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class
from geometric_brownian_motion import geometric_brownian_motion
from jump_diffusion import jump_diffusion
from square_root_diffusion import square_root_diffusion
```

```
# valuation
from valuation_class import valuation_class
from valuation_mcs_european import valuation_mcs_european
from valuation_mcs_american import valuation_mcs_american
```

19.5 延伸阅读

本章中的主题可参考以下图书：

- Glasserman, Paul (2004). *Monte Carlo Methods in Financial Engineering*. New York: Springer.
- Hilpisch, Yves (2015). *Derivatives Analytics with Python*. Chichester, England: Wiley Finance.

本章引用的原创论文：

- Cox, John, Stephen Ross, and Mark Rubinstein (1979). “Option Pricing: A Simplified Approach.” *Journal of Financial Economics*, Vol. 7, No. 3, pp. 229-263.
- Kohler, Michael (2010). “A Review on Regression-Based Monte Carlo Methods for Pricing American Options.” In Luc Devroye et al. (eds.): *Recent Developments in Applied Probability and Statistics* (pp. 37-58). Heidelberg: Physica-Verlag.
- Longstaff, Francis, and Eduardo Schwartz (2001). “Valuing American Options by Simulation: A Simple Least Squares Approach.” *Review of Financial Studies*, Vol. 14, No. 1, pp. 113-147.

第 20 章
投资组合估值

价格是你所付出的，价值是你所得到的。

——沃伦・巴菲特

到目前为止，构建 DX 衍生品分析库的完整方法以及相关的好处读者应该已经相当清楚了。通过将严格依赖蒙特卡洛模拟作为唯一的数值方法，我们几乎已经完成了分析库的模块化工作，如下所示。

折现

dx.constant_short_rate 类的实例负责处理对应的风险中立折现。

相关数据

相关数据、参数和其他输入都保存在 dx.market_environment 类的（多个）实例中。

模拟对象

相关风险因素（标的）的模型是以下 3 个模拟类之一的实例：

- dx.geometric_brownian_motion；
- dx.jump_diffusion；
- dx.square_root_diffusion。

估值对象

需要估值的期权和衍生品的模型是以下两个估值类之一的实例：

- dx.valuation_mcs_european；

- `dx.valuation_mcs_american`。

最后还需要一步：期权和衍生品投资组合（可能很复杂）的估值。为此，我们需要如下要素。

非冗余性

　　每种风险因素（标的）只能建模一次，可能供多个估值对象使用。

相关性

　　必须考虑风险因素之间的相关性。

头寸

　　例如，开仓头寸由一定数量的期权合约组成。

尽管在理论上允许（甚至要求）为模拟和估值对象分别提供币种，但是我们假定所要估值的投资组合仅以单一货币标价。这明显简化了投资组合中价值汇总的工作，因为我们无须考虑汇率和货币风险。

本章介绍两个新类：较为简单的一个类用于建立衍生品头寸模型，较复杂的一个类用于建立衍生品投资组合模型并进行估值。本章的结构如下。

衍生品头寸

　　本节介绍为单一衍生品头寸建模的类。

衍生品投资组合

　　本节介绍投资组合估值（可能包含许多衍生品头寸）的核心类。

20.1　衍生品头寸

理论上，衍生品头寸不过就是用于建立金融工具模型的估值对象和数量的组合。

20.1.1　类

下面的代码展示了建立衍生品头寸模型的类。它主要是存放其他数据和对象的容器。此外，该类提供 `get_info()`方法，用于打印保存在类实例中的数据和对象信息。

```
#
# DX Package
#
# Portfolio -- Derivatives Position Class
#
# derivatives_position.py
#
```

```
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#


class derivatives_position(object):
    ''' Class to model a derivatives position.

    Attributes
    ==========

    name: str
        name of the object
    quantity: float
        number of assets/derivatives making up the position
    underlying: str
        name of asset/risk factor for the derivative
    mar_env: instance of market_environment
        constants, lists, and curves relevant for valuation_class
    otype: str
        valuation class to use
    payoff_func: str
        payoff string for the derivative

    Methods
    =======
    get_info:
        prints information about the derivatives position
    '''

    def __init__(self, name, quantity, underlying, mar_env,
                 otype, payoff_func):
        self.name = name
        self.quantity = quantity
        self.underlying = underlying
        self.mar_env = mar_env
        self.otype = otype
        self.payoff_func = payoff_func
        def get_info(self):
            print('NAME')
            print(self.name, '\n')
            print('QUANTITY')
            print(self.quantity, '\n')
            print('UNDERLYING')
            print(self.underlying, '\n')
```

```
            print('MARKET ENVIRONMENT')
            print('\n**Constants**')
            for key, value in self.mar_env.constants.items():
                print(key, value)
            print('\n**Lists**')
            for key, value in self.mar_env.lists.items():
                print(key, value)
            print('\n**Curves**')
            for key in self.mar_env.curves.items():
                print(key, value)
            print('\nOPTION TYPE')
            print(self.otype, '\n')
            print('PAYOFF FUNCTION')
            print(self.payoff_func)
```

为了定义衍生品头寸，我们需要提供如下信息，这几乎和估值类的实例化相同。

name

表示头寸名称的字符串对象。

quantity

期权/衍生品数量。

underlying

作为风险因素的模拟对象实例。

mar_env

`dx.market_environment` 实例。

otype

字符串，`European` 或者 `American`。

payoff_func

表示收益的 Python 字符串对象。

20.1.2　用例

下面的交互式会话说明了该类的用法。不过，我们必须首先定义一个模拟对象（但不是完整的，只需要最终的对象特性信息）：

```
In [99]: from dx_valuation import *

In [100]: me_gbm = market_environment('me_gbm', dt.datetime(2020, 1, 1)) ❶
```

```
In [101]: me_gbm.add_constant('initial_value', 36.) ❶
          me_gbm.add_constant('volatility', 0.2) ❶
          me_gbm.add_constant('currency', 'EUR') ❶

In [102]: me_gbm.add_constant('model', 'gbm') ❷
```

❶ 表示标的的 `dx.market_environment` 对象。

❷ 此处必须指定模型类型。

类似地，在定义衍生品头寸时，我们不需要完整的 `dx.market_environment` 对象。遗漏的信息在后面（投资组合估值期间）模拟对象初始化时提供：

```
In [103]: from derivatives_position import derivatives_position

In [104]: me_am_put = market_environment('me_am_put', dt.datetime
(2020, 1, 1)) ❶

In [105]: me_am_put.add_constant('maturity', dt.datetime(2020, 12, 31)) ❶
          me_am_put.add_constant('strike', 40.) ❶
          me_am_put.add_constant('currency', 'EUR') ❶

In [106]: payoff_func = 'np.maximum(strike - instrument_values, 0)' ❷

In [107]: am_put_pos = derivatives_position(
                       name='am_put_pos',
                       quantity=3,
                       underlying='gbm',
                       mar_env=me_am_put,
                       otype='American',
                       payoff_func=payoff_func) ❸

In [108]: am_put_pos.get_info()
          NAME
          am_put_pos

          QUANTITY
          3

          UNDERLYING
          gbm

          MARKET ENVIRONMENT

          **Constants**
          maturity 2020-12-31 00:00:00
          strike 40.0
```

```
currency EUR

**Lists**

**Curves**

OPTION TYPE
American

PAYOFF FUNCTION
np.maximum(strike - instrument_values, 0)
```

❶ 表示衍生品的 `dx.market_environment` 对象。

❷ 衍生品的收益函数。

❸ `derivatives_position` 对象实例化。

20.2　衍生品投资组合

从投资组合的角度看，“相关市场”主要是由相关风险因素（标的）及其相关性，以及所要估值的衍生品和衍生品头寸组成的。从理论上，我们现在要处理的是一个一般市场模型 μ（如第 17 章中的定义），并对其应用资产定价基本定理及其推论[1]。

20.2.1　类

下面介绍一个稍复杂的 Python 类，它根据资产定价基本定理实施投资组合估值，考虑了多种相关风险因素和多种衍生品头寸。该类的内嵌文档相当完备，特别是实现特定目的的段落：

```
#
# DX Package
#
# Portfolio -- Derivatives Portfolio Class
#
# derivatives_portfolio.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
```

1　在实践中，我们在此选择的方法有时候称作“总体估值”而不是“特定金融工具估值”。参考 Albanese、Gimonet 和 White（2010a）在 *Risk Magazine* 上发表的论文。——原注

```
import pandas as pd

from dx_valuation import *

# models available for risk factor modeling
models = {'gbm': geometric_brownian_motion,
          'jd': jump_diffusion,
          'srd': square_root_diffusion}

# allowed exercise types
otypes = {'European': valuation_mcs_european,
          'American': valuation_mcs_american}


class derivatives_portfolio(object):
    ''' Class for modeling and valuing portfolios of derivatives positions.

    Attributes
    ==========
    name: str
        name of the object
    positions: dict
        dictionary of positions (instances of derivatives_position class)
    val_env: market_environment
        market environment for the valuation
    assets: dict
        dictionary of market environments for the assets
    correlations: list
        correlations between assets
    fixed_seed: bool
        flag for fixed random number generator seed

    Methods
    =======
    get_positions:
        prints information about the single portfolio positions
    get_statistics:
        returns a pandas DataFrame object with portfolio statistics
    '''

    def __init__(self, name, positions, val_env, assets,
                 correlations=None, fixed_seed=False):
        self.name = name
        self.positions = positions
        self.val_env = val_env
```

```
        self.assets = assets
        self.underlyings = set()
        self.correlations = correlations
        self.time_grid = None
        self.underlying_objects = {}
        self.valuation_objects = {}
        self.fixed_seed = fixed_seed
        self.special_dates = []
        for pos in self.positions:
            # determine earliest starting_date
            self.val_env.constants['starting_date'] = \
                min(self.val_env.constants['starting_date'],
                    positions[pos].mar_env.pricing_date)
            # determine latest date of relevance
            self.val_env.constants['final_date'] = \
                max(self.val_env.constants['final_date'],
                    positions[pos].mar_env.constants['maturity'])
            # collect all underlyings and
            # add to set (avoids redundancy)
            self.underlyings.add(positions[pos].underlying)

        # generate general time grid
        start = self.val_env.constants['starting_date']
        end = self.val_env.constants['final_date']
        time_grid = pd.date_range(start=start, end=end,
                                  freq=self.val_env.constants['frequency']
                                  ).to_pydatetime()
        time_grid = list(time_grid)
        for pos in self.positions:
            maturity_date = positions[pos].mar_env.constants['maturity']
            if maturity_date not in time_grid:
                time_grid.insert(0, maturity_date)
                self.special_dates.append(maturity_date)
        if start not in time_grid:
            time_grid.insert(0, start)
        if end not in time_grid:
            time_grid.append(end)
        # delete duplicate entries
        time_grid = list(set(time_grid))
        # sort dates in time_grid
        time_grid.sort()
        self.time_grid = np.array(time_grid)
        self.val_env.add_list('time_grid', self.time_grid)

        if correlations is not None:
```

```
            # take care of correlations
            ul_list = sorted(self.underlyings)
            correlation_matrix = np.zeros((len(ul_list), len(ul_list)))
            np.fill_diagonal(correlation_matrix, 1.0)
            correlation_matrix = pd.DataFrame(correlation_matrix,
                                              index=ul_list, columns=
ul_list)
            for i, j, corr in correlations:
                corr = min(corr, 0.999999999999)
                # fill correlation matrix
                correlation_matrix.loc[i, j] = corr
                correlation_matrix.loc[j, i] = corr
            # determine Cholesky matrix
            cholesky_matrix = np.linalg.cholesky(np.array(correlation_matrix))
            # dictionary with index positions for the
            # slice of the random number array to be used by
            # respective underlying
            rn_set = {asset: ul_list.index(asset)
                      for asset in self.underlyings}

            # random numbers array, to be used by
            # all underlyings (if correlations exist)
            random_numbers = sn_random_numbers((len(rn_set),
                                               len(self.time_grid),
                                               self.val_env.constants
['paths']),
                                              fixed_seed=self.fixed_seed)

            # add all to valuation environment that is
            # to be shared with every underlying
            self.val_env.add_list('cholesky_matrix', cholesky_matrix)
            self.val_env.add_list('random_numbers', random_numbers)
            self.val_env.add_list('rn_set', rn_set)

        for asset in self.underlyings:
            # select market environment of asset
            mar_env = self.assets[asset]
            # add valuation environment to market environment
            mar_env.add_environment(val_env)
            # select right simulation class
            model = models[mar_env.constants['model']]
            # instantiate simulation object
            if correlations is not None:
                self.underlying_objects[asset] = model(asset, mar_env,
                                                       corr=True)
```

```
            else:
                self.underlying_objects[asset] = model(asset, mar_env,
                                                       corr=False)

        for pos in positions:
            # select right valuation class (European, American)
            val_class = otypes[positions[pos].otype]
            # pick market environment and add valuation environment
            mar_env = positions[pos].mar_env
            mar_env.add_environment(self.val_env)
            # instantiate valuation class
            self.valuation_objects[pos] = \
                val_class(name=positions[pos].name,
                          mar_env=mar_env,
                          underlying=self.underlying_objects[
                    positions[pos].underlying],
                payoff_func=positions[pos].payoff_func)

    def get_positions(self):
        ''' Convenience method to get information about
        all derivatives positions in a portfolio. '''
        for pos in self.positions:
            bar = '\n' + 50 * '-'
            print(bar)
            self.positions[pos].get_info()
            print(bar)

    def get_statistics(self, fixed_seed=False):
        ''' Provides portfolio statistics. '''
        res_list = []
        # iterate over all positions in portfolio
        for pos, value in self.valuation_objects.items():
            p = self.positions[pos]
            pv = value.present_value(fixed_seed=fixed_seed)
            res_list.append([
                p.name,
                p.quantity,
                # calculate all present values for the single instruments
                pv,
                value.currency,
                # single instrument value times quantity
                pv * p.quantity,
                # calculate delta of position
                value.delta() * p.quantity,
                # calculate vega of position
```

```
                value.vega() * p.quantity,
            ])
        # generate a pandas DataFrame object with all results
        res_df = pd.DataFrame(res_list,
                              columns=['name', 'quant.', 'value', 'curr.',
                                       'pos_value', 'pos_delta', 'pos_vega'])
        return res_df
```

> **面向对象**
>
> `dx.derivatives_portfolio`类说明了第6章介绍的面向对象方法的诸多好处。乍一看，它有点像一堆复杂的 Python 代码。但是，它所解决的金融问题相当复杂，并提供了处理大量不同用例的灵活性。很难想象，不使用面向对象编程方法和 Python 类，如何实现这些功能。

20.2.2 用例

按照 DX 分析库的设计概略来看，建模能力被限制于模拟类和估值类的组合。共有 6 种可能的组合：

```
models = {'gbm' : geometric_brownian_motion,
          'jd' : jump_diffusion
          'srd': square_root_diffusion}

otypes = {'European' : valuation_mcs_european,
          'American' : valuation_mcs_american}
```

在下面的交互式用例中，我们组合选择的元素以定义两种不同的衍生品头寸，并将其合并为一个投资组合。

我们以 `derivatives_position` 类和之前的 `gbm` 及 `am_put_pos` 对象为基础构建这个用例。为了说明 `derivatives_position` 类的用法，我们定义一个附加的标的和附加的期权头寸。首先是一个 `dx.jump_diffusion` 对象：

```
In [109]: me_jd = market_environment('me_jd', me_gbm.pricing_date)

In [110]: me_jd.add_constant('lambda', 0.3) ❶
          me_jd.add_constant('mu', -0.75)
          me_jd.add_constant('delta', 0.1)
          me_jd.add_environment(me_gbm) ❷

In [111]: me_jd.add_constant('model', 'jd') ❸
```

❶ 添加跳跃扩散特定参数。

❷ 添加 `gbm` 的其他参数。

❸ 投资组合估值所需的信息。

其次是基于这个新模拟对象的欧式看涨期权：

```
In [112]: me_eur_call = market_environment('me_eur_call', me_jd.pricing_date)

In [113]: me_eur_call.add_constant('maturity', dt.datetime(2020, 6, 30))
          me_eur_call.add_constant('strike', 38.)
          me_eur_call.add_constant('currency', 'EUR')

In [114]: payoff_func = 'np.maximum(maturity_value - strike, 0)'

In [115]: eur_call_pos = derivatives_position(
                      name='eur_call_pos',
                      quantity=5,
                      underlying='jd',
                      mar_env=me_eur_call,
                      otype='European',
                      payoff_func=payoff_func)
```

从投资组合的角度看，现在的相关市场如下面的 `underlyings` 和 `positions` 所示。目前，定义中没有包含标的物之间的相关性。为投资组合估值编辑 `dx.market_environment` 是实例化 `derivatives_portfolio` 类之前的最后一步：

```
In [116]: underlyings = {'gbm': me_gbm, 'jd' : me_jd} ❶
          positions = {'am_put_pos' : am_put_pos,
                       'eur_call_pos' : eur_call_pos} ❷

In [117]: csr = constant_short_rate('csr', 0.06) ❸

In [118]: val_env = market_environment('general', me_gbm.pricing_date)
          val_env.add_constant('frequency', 'W')
          val_env.add_constant('paths', 25000)
          val_env.add_constant('starting_date', val_env.pricing_date)
          val_env.add_constant('final_date', val_env.pricing_date) ❹
          val_env.add_curve('discount_curve', csr) ❸

In [119]: from derivatives_portfolio import derivatives_portfolio

In [120]: portfolio = derivatives_portfolio(
                          name='portfolio',
                          positions=positions,
                          val_env=val_env,
                          assets=underlyings,
                          fixed_seed=False) ❺
```

❶ 相关风险因素。

❷ 相关投资组合头寸。

❸ 投资组合估值独有的折现对象。

❹ `final_date` 未知，因此，将 `pricing_date` 设置为初始值。

❺ 实例化 `derivatives_portfolio` 对象。

现在，我们可以利用估值类的能力来获得刚刚定义的 `derivatives_portfolio` 对象的不同统计量。头寸价值总和、Delta 和 Vega 也很容易计算。这个投资组合的 Delta（几乎中立）和 Vega 值稍微大一些：

```
In [121]: %time portfolio.get_statistics(fixed_seed=False)
          CPU times: user 4.68 s, sys: 409 ms, total: 5.09 s
          Wall time: 14.5 s

Out[121]:
              Name  quant.     value curr.  pos_value  pos_delta  pos_vega
    0   am_put_pos       3 4.458891  EUR  13.376673    -2.0430   31.7850
    1 eur_call_pos       5 2.828634  EUR  14.143170     3.2525   42.2655

In [122]: portfolio.get_statistics(fixed_seed=False)[
              ['pos_value', 'pos_delta', 'pos_vega']].sum() ❶
Out[122]: pos_value    27.502731
          pos_delta     1.233500
          pos_vega     74.050500
          dtype: float64

In [123]: portfolio.get_positions() ❷

          ...

In [124]: portfolio.valuation_objects['am_put_pos'].present_value() ❸
Out[124]: 4.453187

In [125]: portfolio.valuation_objects['eur_call_pos'].delta() ❹
Out[125]: 0.6514
```

❶ 单一头寸价值加总。

❷ 这个方法调用会创建关于所有头寸的冗长输出。

❸ 单一头寸现值估算。

❹ 单一头寸 Dalta 值估算。

这种衍生品投资组合估值根据风险因素不相关的假设进行。这很容易通过检查两条模拟路径（每个模拟对象一条）来进行验证（见图 20-1）：

```
In [126]: path_no = 888
          path_gbm = portfolio.underlying_objects[
             'gbm'].get_instrument_values()[:, path_no]
          path_jd = portfolio.underlying_objects[
             'jd'].get_instrument_values()[:, path_no]

In [127]: plt.figure(figsize=(10,6))
          plt.plot(portfolio.time_grid, path_gbm, 'r', label='gbm')
          plt.plot(portfolio.time_grid, path_jd, 'b', label='jd')
          plt.xticks(rotation=30)
          plt.legend(loc=0)
```

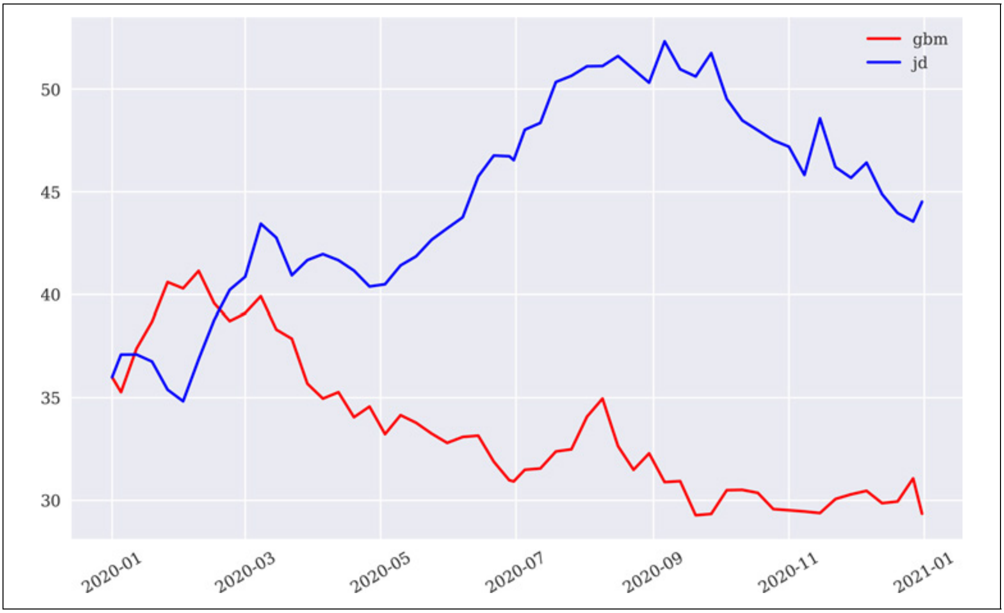


图 20-1　不相关的风险因素

现在考虑两种风险因素高度正相关的情况。在这种情况下，对投资组合中单一头寸没有直接影响：

```
In [128]: correlations = [['gbm', 'jd', 0.9]]

In [129]: port_corr = derivatives_portfolio(
                          name='portfolio',
                          positions=positions,
                          val_env=val_env,
```

```
                              assets=underlyings,
                              correlations=correlations,
                              fixed_seed=True)

In [130]: port_corr.get_statistics()
Out[130]:
                Name  quant.     value curr.  pos_value  pos_delta  pos_vega
     0    am_put_pos       3  4.458556   EUR 13.375668   -2.0376  30.8676
     1  eur_call_pos       5  2.817813   EUR 14.089065    3.3375  42.2340
```

但是，相关性在幕后起到了作用。图 20-2 采用和前面一样的两条路径。这两条路径现在几乎平行移动：

```
In [131]: path_gbm = port_corr.underlying_objects['gbm'].\
                        get_instrument_values()[:, path_no]
          path_jd = port_corr.underlying_objects['jd'].\
                        get_instrument_values()[:, path_no]

In [132]: plt.figure(figsize=(10, 6))
          plt.plot(portfolio.time_grid, path_gbm, 'r', label='gbm')
          plt.plot(portfolio.time_grid, path_jd, 'b', label='jd')
          plt.xticks(rotation=30)
          plt.legend(loc=0);
```

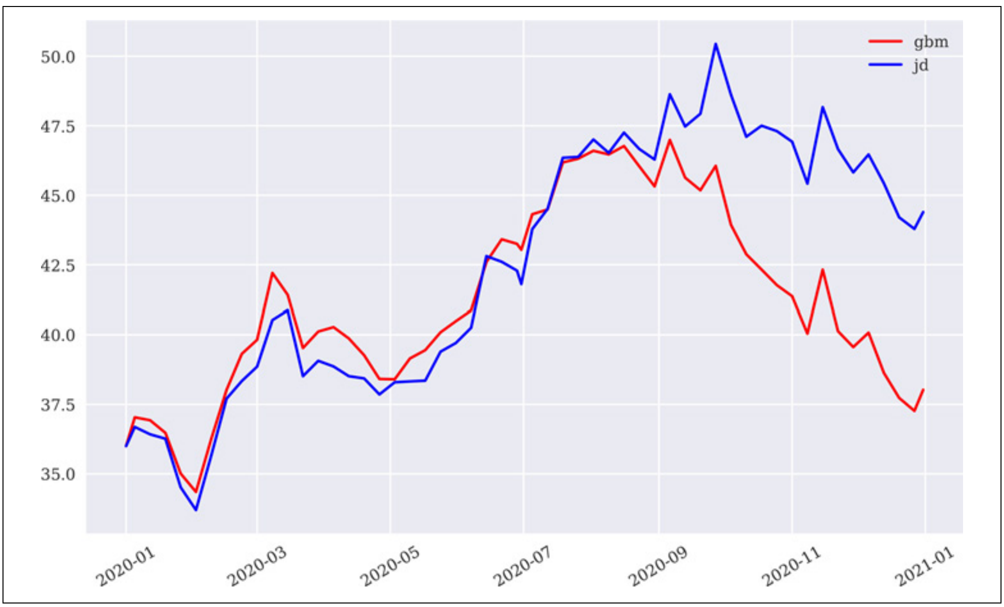


图 20-2　相关风险因素（两条样本路径）

作为最后一个数值化和概念示例，请考虑投资组合现值的频率分布。这是采用其他方

法（如应用分析公式或者二项式期权定价模型）通常不可能生成的。设置参数 `full=True` 可以在现值估算之后按照期权头寸返回完整的一组现值：

```
In [133]: pv1 = 5 * port_corr.valuation_objects['eur_call_pos'].\
                      present_value(full=True)[1]
          pv1
Out[133]: array([ 0.        , 39.71423714, 24.90720272, ..., 0. ,
                 6.42619093,  8.15838265])

In [134]: pv2 = 3 * port_corr.valuation_objects['am_put_pos'].\
                      present_value(full=True)[1]
          pv2
Out[134]: array([21.31806027, 10.71952869, 19.89804376, ..., 21.39292703,
                17.59920608,  0.        ])
```

首先，我们比较两种头寸的频率分布。如图 20-3 所示，两种头寸的收益剖面有很大不同。注意，为了更好的可读性，我们限制了 x 和 y 轴：

```
In [135]: plt.figure(figsize=(10, 6))
          plt.hist([pv1, pv2], bins=25,
                   label=['European call', 'American put']);
          plt.axvline(pv1.mean(), color='r', ls='dashed',
                      lw=1.5, label='call mean = %4.2f' % pv1.mean())
          plt.axvline(pv2.mean(), color='r', ls='dotted',
                      lw=1.5, label='put mean = %4.2f' % pv2.mean())
          plt.xlim(0, 80); plt.ylim(0, 10000)
          plt.legend();
```

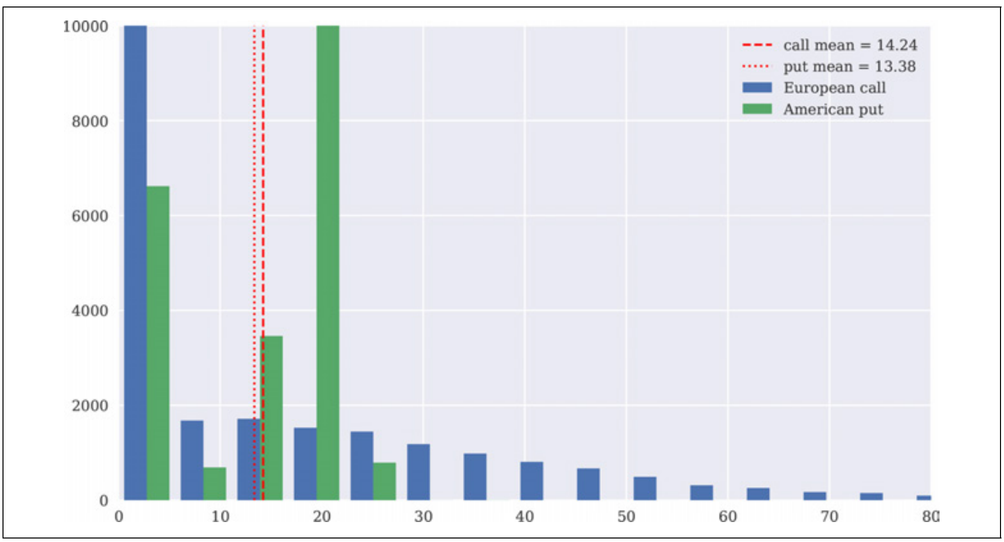


图 20-3　两种头寸现值的频率分布

图 20-4 显示了投资组合现值的全部频率分布，可以清晰地看到组合看涨和看跌期权的多样化对冲效应：

```
In [136]: pvs = pv1 + pv2
          plt.figure(figsize=(10, 6))
          plt.hist(pvs, bins=50, label='portfolio');
          plt.axvline(pvs.mean(), color='r', ls='dashed',
                      lw=1.5, label='mean = %4.2f' % pvs.mean())
          plt.xlim(0, 80); plt.ylim(0, 7000)
          plt.legend();
```

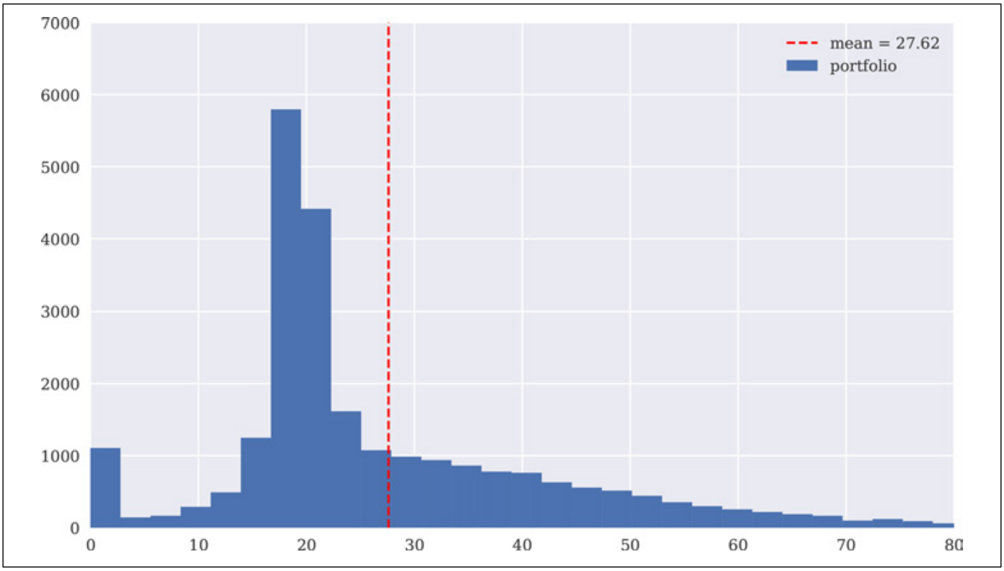


图 20-4　投资组合现值频率分布

以限制标准差为度量指标，两种风险因素之间的相关性对投资组合的风险有何影响？这个问题可以用以下两种估算来回答：

```
In [137]: pvs.std() ❶
Out[137]: 16.723724772741118

In [138]: pv1 = (5 * portfolio.valuation_objects['eur_call_pos'].
                      present_value(full=True)[1])
          pv2 = (3 * portfolio.valuation_objects['am_put_pos'].
                      present_value(full=True)[1])
          (pv1 + pv2).std() ❷
Out[138]: 21.80498672323975
```

❶ 有相关投资组合价值标准差。

❷ 无相关投资组合价值标准差。

虽然均值保持恒定（忽略数值化误差），但在这种计量方式下，相关性明显降低了投资组合风险。同样，这是使用其他数值化方法或者估值方法不可能得到的认识。

20.3 结语

本章介绍衍生品头寸投资组合的估值和风险管理，这种投资组合依赖多种可能相关的风险因素。为此，我们引入了一个新类 `derivatives_position` 以建立期权/衍生品头寸的模型。但是，主要的焦点要放在 `derivatives_portfolio` 类，该类实现了一些相当复杂的任务。该类负责以下工作：

- 风险因素之间的相关性（该类为所有风险因素的模拟生成一组一致性的随机数）；
- 在单一市场环境和一般估值环境以及衍生品头寸下的模拟对象实例化；
- 根据所有假设、涉及风险因素和衍生品头寸生成投资组合统计量。

本章介绍的例子只能展示可由 DX 库和 `derivatives_portfolio` 类管理和估值的简单衍生品投资组合。对 DX 库的自然扩展应该添加更复杂的金融模型，例如随机波动率模型，并添加多重风险估值类以建立依赖多种风险因素的衍生品（如欧洲一揽子期权或者美国最大看涨期权）模型并为之估值。在这一阶段，和资产定价基本定理（"全局估值"）一样通用的模块化建模以及估值框架应用展现出优势：风险因素的非冗余建模以及对风险因素之间相关性的考虑，也影响着多风险衍生品的价值和"希腊字母"。

下面是最终的包装器模块，在单条 `import` 语句中组合了 DX 分析库的所有组件：

```
#
# DX Package
#
# All components
#
# dx_package.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
from dx_valuation import *
from derivatives_position import derivatives_position
from derivatives_portfolio import derivatives_portfolio
```

下面是 dx 文件夹中`__init__.py` 文件的完整内容：

```
#
# DX Package
```

```
# packaging file
# __init__.py
#
import numpy as np
import pandas as pd
import datetime as dt

# frame
from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment
from plot_option_stats import plot_option_stats

# simulation
from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class
from geometric_brownian_motion import geometric_brownian_motion
from jump_diffusion import jump_diffusion
from square_root_diffusion import square_root_diffusion

# valuation
from valuation_class import valuation_class
from valuation_mcs_european import valuation_mcs_european
from valuation_mcs_american import valuation_mcs_american

# portfolio
from derivatives_position import derivatives_position
from derivatives_portfolio import derivatives_portfolio
```

20.4 延伸阅读

和前面关于 DX 衍生品分析库的内容一样，Glasserman（2004）提供了金融工程和应用中关于蒙特卡洛模拟的比较全面的分析。Hilpisch（2015）也提供了非常重要的蒙特卡洛算法基于 Python 的实现：

- Glasserman, Paul (2004). *Monte Carlo Methods in Financial Engineering*. New York: Springer.
- Hilpisch, Yves (2015). *Derivatives Analytics with Python*. Chichester, England: Wiley Finance.

但是，利用蒙特卡洛模拟，以一致、无冗余的方式估算（复杂）衍生品投资组合价值方面没有太多研究。值得注意的一个例外（至少从概念上讲）是 Albanese、Gimonet

和 White（2010a）的简短论文。这几位作者组成的同一团队所撰写的白皮书更详细一些：

- Albanese, Claudio, Guillaume Gimonet and Steve White (2010a). “Towards a Global Valuation Model”. *Risk Magazine*, Vol. 23, No. 5, pp. 68-71.
- Albanese, Claudio, Guillaume Gimonet and Steve White (2010b). “Global Valuation and Dynamic Risk Management”. Working paper.

第 21 章

基于市场的估值

我们正面对着极端的波动性。

——卡洛斯・戈恩

基于市场的非流动性交易期权及衍生品估值是衍生品分析的一项重要任务。为此，人们通常要根据流动性交易期权的市场报价验证定价模型，然后使用经过验证的模型为非交易期权定价。[1]

本章介绍基于 DX 库的一个案例，说明这个在前 4 章中逐步开发的库适合于实施基于市场的估值。本案例以 DAX30 股票指数为基础，该指数是由 30 家德国大公司股票组成的蓝筹股市场指数。市场上有以这个指数为标的的流动交易欧式看涨及看跌期权。本章分成以下几个部分。

期权数据

我们需要两类数据，即 DAX30 股票指数本身的数据，以及该指数的流动交易欧式期权数据。

模型检验

为了以和市场一致的方式估算非交易期权，通常首先用期权报价检验所选择的模型，证明基于最优参数的模型会尽可能地复制市场价格。

投资组合估值

有了上述的所有数据和经过市场检验的 DAX 30 指数模型后，最后一项任务是建立非交易权证的模型并为之估值；还要根据头寸和投资规模估算重要风险测度。

1　详见 Hilpisch（2015）。——原注

本章使用的指数与期权数据来自 Thomson Reuters Eikon Data API（参见 21.4 节）。

21.1 期权数据

首先是必要的导入与设置：

```
In [1]: import numpy as np
        import pandas as pd
        import datetime as dt

In [2]: from pylab import mpl, plt
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline

In [3]: import sys
        sys.path.append('../')
        sys.path.append('../dx')
```

利用 21.4 节中创建的数据文件，用 pandas 读取期权数据，并将其中的日期信息处理为 `pd.Timestamp` 对象：

```
In [4]: dax = pd.read_csv('../../source/tr_eikon_option_data.csv',
                          index_col=0) ❶
In [5]: for col in ['CF_DATE', 'EXPIR_DATE']:
            dax[col] = dax[col].apply(lambda date: pd.Timestamp(date)) ❷

In [6]: dax.info() ❸
        <class 'pandas.core.frame.DataFrame'>
        Int64Index: 115 entries, 0 to 114
        Data columns (total 7 columns):
        Instrument    115 non-null object
        CF_DATE       115 non-null datetime64[ns]
        EXPIR_DATE    114 non-null datetime64[ns]
        PUTCALLIND    114 non-null object
        STRIKE_PRC    114 non-null float64
        CF_CLOSE      115 non-null float64
        IMP_VOLT      114 non-null float64
        dtypes: datetime64[ns](2), float64(3), object(2)
        memory usage: 7.2+ KB

In [7]: dax.set_index('Instrument').head(7) ❸
Out[7]:
                     CF_DATE EXPIR_DATE PUTCALLIND STRIKE_PRC CF_CLOSE \
Instrument
```

```
.GDAXI           2018-04-27        NaT         NaN       NaN  12500.47
GDAX105000G8.EX 2018-04-27 2018-07-20        CALL   10500.0   2040.80
GDAX105000S8.EX 2018-04-27 2018-07-20        PUT    10500.0     32.00
GDAX108000G8.EX 2018-04-27 2018-07-20        CALL   10800.0   1752.40
GDAX108000S8.EX 2018-04-26 2018-07-20        PUT    10800.0     43.80
GDAX110000G8.EX 2018-04-27 2018-07-20        CALL   11000.0   1562.80
GDAX110000S8.EX 2018-04-27 2018-07-20        PUT    11000.0     54.50

                 IMP_VOLT
Instrument
.GDAXI                NaN
GDAX105000G8.EX     23.59
GDAX105000S8.EX     23.59
GDAX108000G8.EX     22.02
GDAX108000S8.EX     22.02
GDAX110000G8.EX     21.00
GDAX110000S8.EX     21.00
```

❶ 用 pd.read_csv()读取数据。

❷ 处理包含数据信息的两列。

❸ 得到的 DataFrame 对象。

下面的代码在一个变量中保存 DAX 30 相关指数水平，并创建两个新的 DataFrame 对象，一个用于看涨期权，另一个用于看跌期权。图 21-1 显示了看涨期权的市场报价以及隐含波动率：[1]

```
In [8]: initial_value = dax.iloc[0]['CF_CLOSE'] ❶

In [9]: calls = dax[dax['PUTCALLIND'] == 'CALL'].copy() ❷
        puts = dax[dax['PUTCALLIND'] == 'PUT '].copy() ❷

In [10]: calls.set_index('STRIKE_PRC')[['CF_CLOSE', 'IMP_VOLT']].plot(
             secondary_y='IMP_VOLT', style=['bo', 'rv'], figsize=(10, 6));
```

❶ 将相关指数水平赋给 initial_value 变量。

❷ 将看涨与看跌期权数据分到两个新的 DataFrame 对象中。

图 21-2 显示看跌期权的市场报价及隐含波动率：

```
In [11]: ax = puts.set_index('STRIKE_PRC')[['CF_CLOSE', 'IMP_VOLT']].plot(
             secondary_y='IMP_VOLT', style=['bo', 'rv'], figsize=(10, 6))
         ax.get_legend().set_bbox_to_anchor((0.25, 0.5));
```

1　期权的隐含波动率是在其他条件不变的情况下，用 Black-Scholes-Merton(1973)期权定价公式得出的期权市场报价的波动率值。——原注

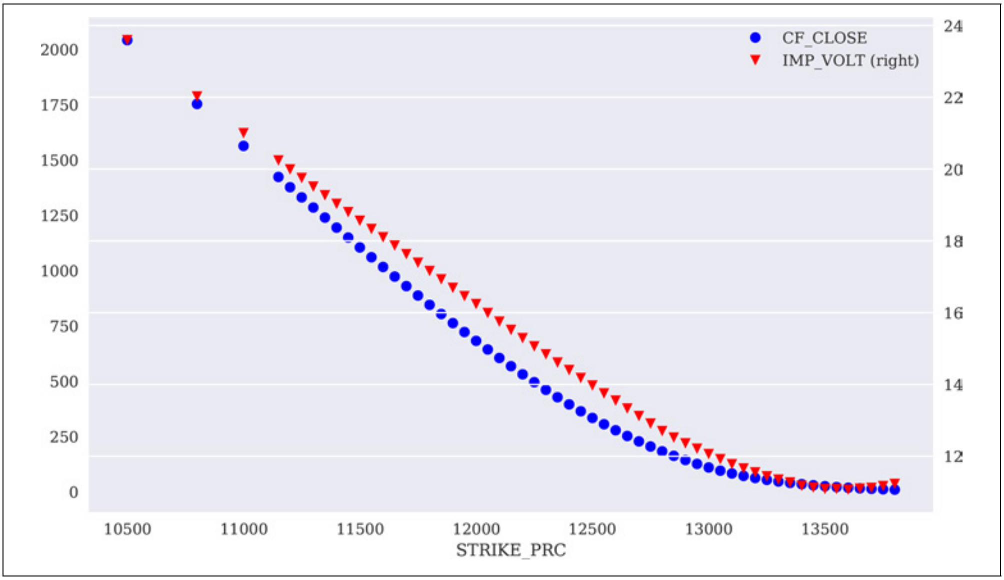


图 21-1　DAX 30 欧式看涨期权市场报价与隐含波动率

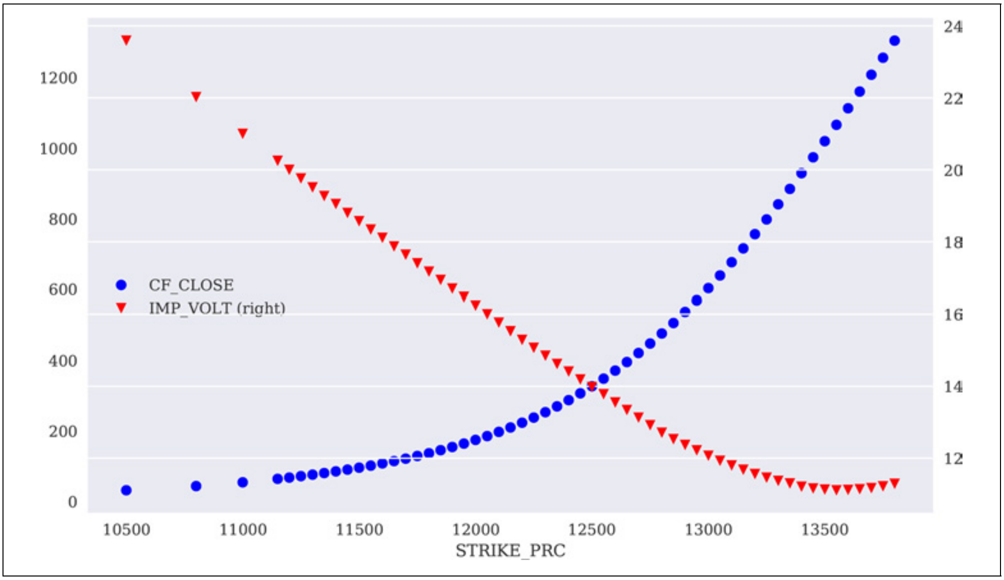


图 21-2　DAX 30 欧式看跌期权市场报价与隐含波动率

21.2　模型检验

本节选择相关的市场数据来建立 DAX 30 指数欧式期权模型，并检验本身的规程。

21.2.1 相关市场数据

模型检验通常在可用期权市场报价的较小子集上进行。[1]为此，以下代码只选择行权价相对接近当前指数水平的欧式看涨期权（见图 21-3）。换言之，只选择差价不太大的价内或者价外欧式看涨期权：

```
In [12]: limit = 500 ❶

In [13]: option_selection = calls[abs(calls['STRIKE_PRC'] - initial_value)
                                   < limit].copy() ❷

In [14]: option_selection.info() ❸
         <class 'pandas.core.frame.DataFrame'>
         Int64Index: 20 entries, 43 to 81
         Data columns (total 7 columns):
         Instrument    20 non-null object
         CF_DATE       20 non-null datetime64[ns]
         EXPIR_DATE    20 non-null datetime64[ns]
         PUTCALLIND    20 non-null object
         STRIKE_PRC    20 non-null float64
         CF_CLOSE      20 non-null float64
         IMP_VOLT      20 non-null float64
         dtypes: datetime64[ns](2), float64(3), object(2)
         memory usage: 1.2+ KB

In [15]: option_selection.set_index('Instrument').tail() ❸
Out[15]:
                       CF_DATE EXPIR_DATE PUTCALLIND STRIKE_PRC CF_CLOSE \
    Instrument
    GDAX128000G8.EX 2018-04-27 2018-07-20    CALL    12800.0   182.4
    GDAX128500G8.EX 2018-04-27 2018-07-20    CALL    12850.0   162.0
    GDAX129000G8.EX 2018-04-25 2018-07-20    CALL    12900.0   142.9
    GDAX129500G8.EX 2018-04-27 2018-07-20    CALL    12950.0   125.4
    GDAX130000G8.EX 2018-04-27 2018-07-20    CALL    13000.0   109.4

                     IMP_VOLT
    Instrument
    GDAX128000G8.EX     12.70
    GDAX128500G8.EX     12.52
    GDAX129000G8.EX     12.36
    GDAX129500G8.EX     12.21
    GDAX130000G8.EX     12.06
```

1　详见 Hilpisch (2015)的第 11 章。——原注

```
In [16]: option_selection.set_index('STRIKE_PRC')[['CF_CLOSE', 'IMP_
VOLT']].plot(
                secondary_y='IMP_VOLT', style=['bo', 'rv'], figsize=(10, 6));
```

❶ 设置 `limit` 值，表示行权价与当前指数水平的偏差范围（货币性条件）。

❷ 根据 `limit` 值选择包含在检验中的欧式看涨期权。

❸ 包含用于检验的欧式看涨期权的 `DataFrame` 对象。

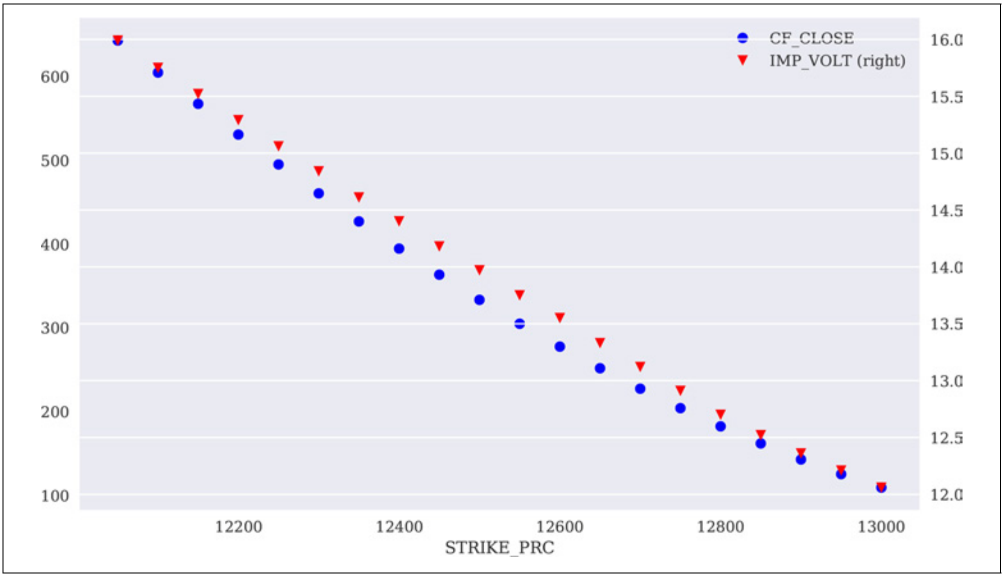


图 21-3　用于模型检验的 DAX 30 欧式看涨期权

21.2.2　期权建模

定义了相关市场数据后，我们现在可以用 DX 库建立欧式看涨期权的模型了。下面是建立 DAX 30 指数模型的 `dx.market_environment` 对象定义，与前面几章的例子相似：

```
In [17]: import dx

In [18]: pricing_date = option_selection['CF_DATE'].max() ❶

In [19]: me_dax = dx.market_environment('DAX30', pricing_date) ❷

In [20]: maturity = pd.Timestamp(calls.iloc[0]['EXPIR_DATE']) ❸

In [21]: me_dax.add_constant('initial_value', initial_value) ❹
         me_dax.add_constant('final_date', maturity) ❹
         me_dax.add_constant('currency', 'EUR') ❹
```

```
In [22]: me_dax.add_constant('frequency', 'B') ❺
         me_dax.add_constant('paths', 10000) ❺

In [23]: csr = dx.constant_short_rate('csr', 0.01) ❻
         me_dax.add_curve('discount_curve', csr) ❻
```

❶ 按照期权数据定义初始值或者定价日期。

❷ 实例化 `dx.market_environment` 对象。

❸ 按照期权数据定义到期日。

❹ 添加模型基本参数。

❺ 添加模拟相关参数。

❻ 定义并添加一个 `dx.constant_short_rate` 对象。

然后，以下代码为 `dx.jump_diffusion` 类添加模型特定参数，并实例化对应的模拟对象：

```
In [24]: me_dax.add_constant('volatility', 0.2)
         me_dax.add_constant('lambda', 0.8)
         me_dax.add_constant('mu', -0.2)
         me_dax.add_constant('delta', 0.1)

In [25]: dax_model = dx.jump_diffusion('dax_model', me_dax)
```

以欧式看涨期权为例，考虑如下参数，行权价设置为与 DAX 30 指数水平相等的值。这就可以进行基于蒙特卡洛模拟的第一次估值了：

```
In [26]: me_dax.add_constant('strike', initial_value) ❶
         me_dax.add_constant('maturity', maturity)

In [27]: payoff_func = 'np.maximum(maturity_value - strike, 0)' ❷

In [28]: dax_eur_call = dx.valuation_mcs_european('dax_eur_call',
                                 dax_model, me_dax, payoff_func) ❸

In [29]: dax_eur_call.present_value() ❹
Out[29]: 654.298085
```

❶ 让 `strike` 值与 `initial_value` 的值相等。

❷ 为欧式看涨期权定义收益函数。

❸ 实例化估值对象。

❹ 实例化模拟与估值。

类似地，我们可以为所有相关的 DAX 30 指数欧式看涨期权定义估值对象。参数中唯一改变的是行权价：

```
In [30]: option_models = {} ❶
         for option in option_selection.index:
             strike = option_selection['STRIKE_PRC'].loc[option] ❷
             me_dax.add_constant('strike', strike) ❷
             option_models[strike] = dx.valuation_mcs_european(
                                         'eur_call_%d' % strike,
                                         dax_model,
                                         me_dax,
                                         payoff_func)
```

❶ 估值对象被收集在一个字典对象中。

❷ 选择相关行权价，在 `dx.market_environment` 对象中（重新）定义。

现在，以所有相关期权的估值对象为基础，`calculate_model_values()`函数取得了一组模型特定参数值 `p0`，并返回所有期权的模型值：

```
In [32]: def calculate_model_values(p0):
             ''' Returns all relevant option values.

             Parameters
             ===========
             p0: tuple/list
                 tuple of kappa, theta, volatility

             Returns
             =======
             model_values: dict
                 dictionary with model values
             '''
             volatility, lamb, mu, delta = p0
             dax_model.update(volatility=volatility, lamb=lamb,
                              mu=mu, delta=delta)
             return {
                    strike: model.present_value(fixed_seed=True)
                    for strike, model in option_models.items()
                 }

In [33]: calculate_model_values((0.1, 0.1, -0.4, 0.0))
Out[33]: {12050.0: 611.222524,
          12100.0: 571.83659,
          12150.0: 533.595853,
          12200.0: 496.607225,
          12250.0: 460.863233,
```

```
            12300.0: 426.543355,
            12350.0: 393.626483,
            12400.0: 362.066869,
            12450.0: 331.877733,
            12500.0: 303.133596,
            12550.0: 275.987049,
            12600.0: 250.504646,
            12650.0: 226.687523,
            12700.0: 204.550609,
            12750.0: 184.020514,
            12800.0: 164.945082,
            12850.0: 147.249829,
            12900.0: 130.831722,
            12950.0: 115.681449,
            13000.0: 101.917351}
```

如下所述，在检验过程中我们使用了 `calculate_model_values()`函数。

21.2.3 检验过程

期权定价模型的检验通常是一个凸优化问题。广为使用的检验函数（最小化某些误差函数值）是给定期权报价下模型期权价值的均方差（MSE）。[1]假定有 N 个相关期权、模型和市场报价，则以 MSE 为基础、用市场报价检验金融模型的问题可以用公式 21-1 表示。式中，C_n^* 和 C_n^{mod} 是第 n 个期权的市场价格和模型价格。p 是作为期权定价模型输入的参数集。

公式 21-1 根据均方差检验模型

$$\min_p \frac{1}{N}\sum_{n=1}^{N}(C_n^* - C_n^{mod}(p))^2$$

Python 函数 `mean_squared_error()`从技术上实现了这种模型检验方法，该函数使用全局变量 `i` 来控制中间参数元组对象和结果 MSE 的输出：

```
In [34]: i = 0
         def mean_squared_error(p0):
             ''' Returns the mean-squared error given
             the model and market values.

             Parameters
             ===========
             p0: tuple/list
                 tuple of kappa, theta, volatility
```

1 定义检验过程的目标函数有多种替代方法。这个主题的讨论参见 Hilpish(2015)第 11 章。——原注

```
        Returns
        =======
        MSE: float
            mean-squared error
        '''
        global i
        model_values = np.array(list(
                calculate_model_values(p0).values()))❶
        market_values = option_selection['CF_CLOSE'].values❷
        option_diffs = model_values - market_values❸
        MSE = np.sum(option_diffs ** 2) / len(option_diffs)❹
        if i % 75 == 0:
            if i == 0:
                print('%4s %6s %6s %6s %6s --> %6s' %
                      ('i', 'vola', 'lambda', 'mu', 'delta', 'MSE'))
            print('%4d %6.3f %6.3f %6.3f %6.3f --> %6.3f' %
                      (i, p0[0], p0[1], p0[2], p0[3], MSE))
        i += 1
        return MSE

In [35]: mean_squared_error((0.1, 0.1, -0.4, 0.0)) ❺
            i    vola  lambda     mu   delta -->     MSE
            0   0.100   0.100 -0.400   0.000 --> 728.375

Out[35]: 728.3752973715275
```

❶ 估算一组模型价值。

❷ 选出市场报价。

❸ 逐个元素地计算两者之间的差值。

❹ 计算误差值的均方根。

❺ 根据样本参数表示这种计算。

第 11 章介绍了用于实现检验过程的两个函数（`spo.brute()`和 `spo.fmin()`）。首先，根据 4 个模型特定参数值的范围进行全局最小化。结果是在暴力最小化期间检查的所有参数组合中最优的那个：

```
In [36]: import scipy.optimize as spo

In [37]: %%time
         i = 0
         opt_global = spo.brute(mean_squared_error,
                        ((0.10, 0.201, 0.025), # range for volatility
                         (0.10, 0.80, 0.10), # range for jump intensity
```

```
                              (-0.40, 0.01, 0.10), # range for average jump size
                              (0.00, 0.121, 0.02)), # range for jump variability
                            finish=None)
           i    vola  lambda     mu  delta -->     MSE
           0   0.100   0.100 -0.400  0.000 --> 728.375
          75   0.100   0.300 -0.400  0.080 --> 5157.513
         150   0.100   0.500 -0.300  0.040 --> 12199.386
         225   0.100   0.700 -0.200  0.000 --> 6904.932
         300   0.125   0.200 -0.200  0.100 --> 855.412
         375   0.125   0.400 -0.100  0.060 --> 621.800
         450   0.125   0.600  0.000  0.020 --> 544.137
         525   0.150   0.100  0.000  0.120 --> 3410.776
         600   0.150   0.400 -0.400  0.080 --> 46775.769
         675   0.150   0.600 -0.300  0.040 --> 56331.321
         750   0.175   0.100 -0.200  0.000 --> 14562.213
         825   0.175   0.300 -0.200  0.100 --> 24599.738
         900   0.175   0.500 -0.100  0.060 --> 19183.167
         975   0.175   0.700  0.000  0.020 --> 11871.683
        1050   0.200   0.200  0.000  0.120 --> 31736.403
        1125   0.200   0.500 -0.400  0.080 --> 130372.718
        1200   0.200   0.700 -0.300  0.040 --> 126365.140
        CPU times: user 1min 45s, sys: 7.07 s, total: 1min 52s
        Wall time: 1min 56s

In [38]: mean_squared_error(opt_global)
Out[38]: 17.946670038040985
```

opt_global 值只是中间结果，是局部最小化的起始值。在给定的参数下，opt_global 值是假定容限下的最终和最优值：

```
In [39]: %%time
         i = 0
         opt_local = spo.fmin(mean_squared_error, opt_global,
                              xtol=0.00001, ftol=0.00001,
                              maxiter=200, maxfun=550)
            i   vola  lambda      mu   delta -->    MSE
            0  0.100   0.200  -0.300   0.000 --> 17.947
           75  0.098   0.216  -0.302  -0.001 -->  7.885
          150  0.098   0.216  -0.300  -0.001 -->  7.371
         Optimization terminated successfully.
                  Current function value: 7.371163
                  Iterations: 100
                  Function evaluations: 188
         CPU times: user 15.6 s, sys: 1.03 s, total: 16.6 s
         Wall time: 16.7 s
```

```
In [40]: i = 0
         mean_squared_error(opt_local) ❶
            i    vola   lambda      mu  delta -->   MSE
            0   0.098   0.216  -0.300 -0.001 --> 7.371

Out[40]: 7.371162645265256

In [41]: calculate_model_values(opt_local) ❷
Out[41]: {12050.0: 647.428189,
          12100.0: 607.402796,
          12150.0: 568.46137,
          12200.0: 530.703659,
          12250.0: 494.093839,
          12300.0: 458.718401,
          12350.0: 424.650128,
          12400.0: 392.023241,
          12450.0: 360.728543,
          12500.0: 330.727256,
          12550.0: 302.117223,
          12600.0: 274.98474,
          12650.0: 249.501807,
          12700.0: 225.678695,
          12750.0: 203.490065,
          12800.0: 182.947468,
          12850.0: 163.907583,
          12900.0: 146.259349,
          12950.0: 129.909743,
          13000.0: 114.852425}
```

❶ 最优参数值下的误差均方根。

❷ 给定最优参数下的模型价值。

接下来，我们比较最优参数模型价值与市场报价，计算模型价值与市场报价之间的绝对差值，并得出它与市场价值的差异比例：

```
In [42]: option_selection['MODEL'] = np.array(list(calculate_model_values(
                                                opt_local).values()))
         option_selection['ERRORS_EUR'] = (option_selection['MODEL'] -
                                          option_selection['CF_CLOSE'])
         option_selection['ERRORS_%'] = (option_selection['ERRORS_EUR'] /
                                        option_selection['CF_CLOSE']) * 100

In [43]: option_selection[['MODEL', 'CF_CLOSE', 'ERRORS_EUR', 'ERRORS_%']]
Out[43]:          MODEL CF_CLOSE  ERRORS_EUR ERRORS_%
         43  647.428189    642.6    4.828189  0.751352
```

```
45  607.402796    604.4   3.002796  0.496823
47  568.461370    567.1   1.361370  0.240058
49  530.703659    530.4   0.303659  0.057251
51  494.093839    494.8  -0.706161 -0.142716
53  458.718401    460.3  -1.581599 -0.343602
55  424.650128    426.8  -2.149872 -0.503719
57  392.023241    394.4  -2.376759 -0.602627
59  360.728543    363.3  -2.571457 -0.707805
61  330.727256    333.3  -2.572744 -0.771900
63  302.117223    304.8  -2.682777 -0.880176
65  274.984740    277.5  -2.515260 -0.906400
67  249.501807    251.7  -2.198193 -0.873338
69  225.678695    227.3  -1.621305 -0.713289
71  203.490065    204.1  -0.609935 -0.298841
73  182.947468    182.4   0.547468  0.300147
75  163.907583    162.0   1.907583  1.177520
77  146.259349    142.9   3.359349  2.350839
79  129.909743    125.4   4.509743  3.596286
81  114.852425    109.4   5.452425  4.983935

In [44]: round(option_selection['ERRORS_EUR'].mean(), 3) ❶
Out[44]: 0.184

In [45]: round(option_selection['ERRORS_%'].mean(), 3) ❷
Out[45]: 0.36
```

❶ 以欧元表示的平均价差。

❷ 以百分比表示的平均价差。

图 21-4 可视化了估值结果与误差：

```
In [46]: fix, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, figsize=(10, 10))
         strikes = option_selection['STRIKE_PRC'].values
         ax1.plot(strikes, option_selection['CF_CLOSE'], label='market quotes')
         ax1.plot(strikes, option_selection['MODEL'], 'ro', label='
model values')
         ax1.set_ylabel('option values')
         ax1.legend(loc=0)
         wi = 15
         ax2.bar(strikes - wi / 2., option_selection['ERRORS_EUR'], width=wi)
         ax2.set_ylabel('errors [EUR]')
         ax3.bar(strikes - wi / 2., option_selection['ERRORS_%'], width=wi)
         ax3.set_ylabel('errors [%]')
         ax3.set_xlabel('strikes');
```

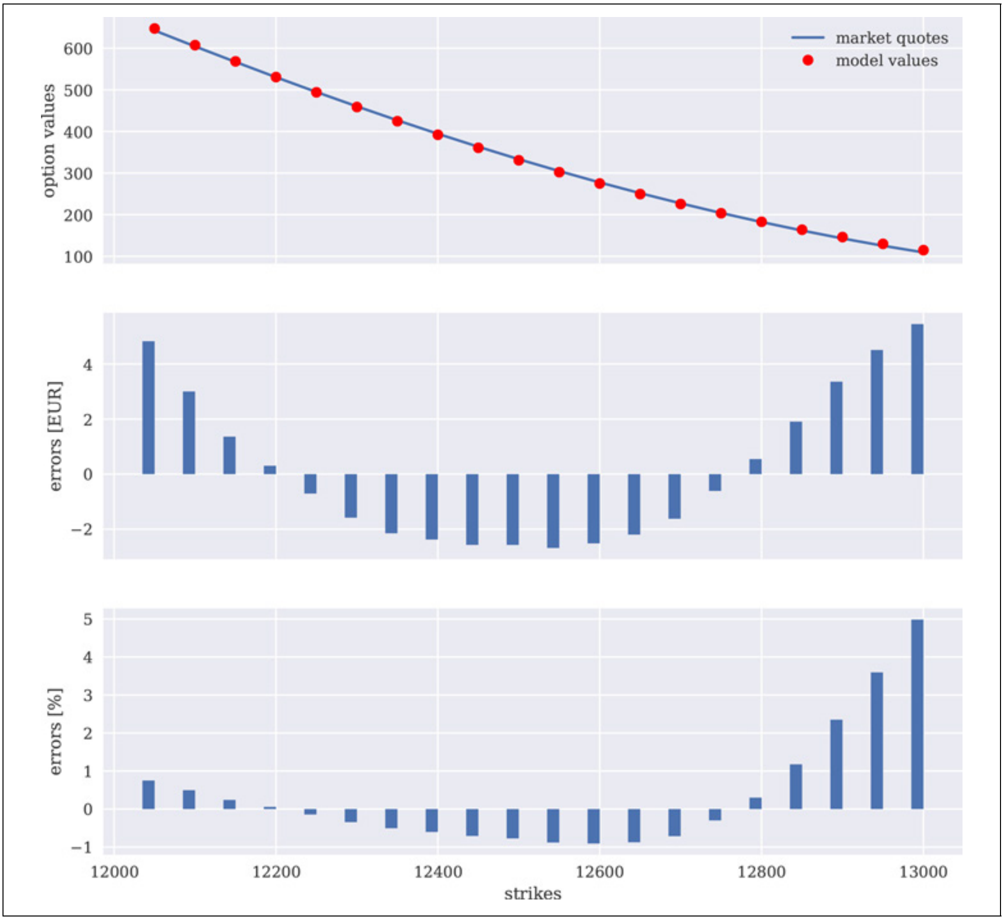


图 21-4　检验后的模型价值和市场报价



检验速度

按照市场数据检验期权定价模型，通常需要重新计算数百个甚至数千个期权价值。因此，计算通常是根据分析定价公式进行的。这里，检验过程依赖蒙特卡洛模拟作为定价方法，该方法在计算要求上高于分析方法。不过，检验过程即便在典型的笔记本电脑上也不会“花费太长时间”。例如，使用并行化技术可以显著加快检验速度。

21.3　投资组合估值

配备了反映金融市场现实、经过检验的模型（由流动交易期权表示）后，我们就可以

建立非交易期权及衍生品的模型并为其估值了。思路是检验通过最优参数，在模型中“注入”了正确的风险中立鞅测度。根据这一测度，我们可以将资产定价基本定理的机制应用到其他（未用于检验的）未定权益上。

本节考虑 DAX 30 指数美式看跌期权的一个投资组合。交易所没有连续进行此类期权的交易。简单起见，假定美式看跌期权的到期日与用于检验的欧式看涨期权相同，并假定行权价也一样。

21.3.1 建立期权头寸模型

首先，使用检验所用的最优参数来建立标的风险因素（DAX 30 指数）市场环境的模型：

```
In [47]: me_dax = dx.market_environment('me_dax', pricing_date)
         me_dax.add_constant('initial_value', initial_value)
         me_dax.add_constant('final_date', pricing_date)
         me_dax.add_constant('currency', 'EUR')

In [48]: me_dax.add_constant('volatility', opt_local[0]) ❶
         me_dax.add_constant('lambda', opt_local[1]) ❶
         me_dax.add_constant('mu', opt_local[2]) ❶
         me_dax.add_constant('delta', opt_local[3]) ❶

In [49]: me_dax.add_constant('model', 'jd')
```

❶ 添加来自检验的最优参数。

其次，定义期权头寸和相关环境，并将它们保存在两个单独的字典对象中：

```
In [50]: payoff_func = 'np.maximum(strike - instrument_values, 0)'

In [51]: shared = dx.market_environment('share', pricing_date) ❶
         shared.add_constant('maturity', maturity) ❶
         shared.add_constant('currency', 'EUR') ❶

In [52]: option_positions = {}
         option_environments = {}
         for option in option_selection.index:
             option_environments[option] = dx.market_environment(
                 'am_put_%d' % option, pricing_date) ❷
             strike = option_selection['STRIKE_PRC'].loc[option] ❸
             option_environments[option].add_constant('strike', strike) ❸
             option_environments[option].add_environment(shared) ❹
             option_positions['am_put_%d' % strike] = \
                             dx.derivatives_position(
                                 'am_put_%d' % strike,
                                 quantity=np.random.randint(10, 50),
```

```
                                  underlying='dax_model',
                                  mar_env=option_environments[option],
                                  otype='American',
                                  payoff_func=payoff_func) ❺
```

❶ 定义共享的 dx.market_environment 对象，将其作为所有期权特定环境的基础。

❷ 为相关美式看跌期权定义并保存一个新的 dx.market_environment 对象。

❸ 为期权定义和保存行权价参数。

❹ 将来自共享的 dx.market_environment 对象的元素添加到期权特定的对象中。

❺ 以随机数定义 dx.derivatives_position 对象。

21.3.2 期权投资组合

为了给所有美式看跌权证的投资组合估值，我们需要一个估值环境。它包含估算头寸价值和风险统计量的主要参数：

```
In [53]: val_env = dx.market_environment('val_env', pricing_date)
         val_env.add_constant('starting_date', pricing_date)
         val_env.add_constant('final_date', pricing_date) ❶
         val_env.add_curve('discount_curve', csr)
         val_env.add_constant('frequency', 'B')
         val_env.add_constant('paths', 25000)

In [54]: underlyings = {'dax_model' : me_dax} ❷

In [55]: portfolio = dx.derivatives_portfolio('portfolio', option_positions,
                                              val_env, underlyings) ❸

In [56]: %time results = portfolio.get_statistics(fixed_seed=True)
         CPU times: user 1min 5s, sys: 2.91 s, total: 1min 8s
         Wall time: 38.2 s

In [57]: results.round(1)
Out[57]:            name  quant.  value curr.  pos_value  pos_delta  pos_vega
         0  am_put_12050      33  151.6  EUR     5002.8       -4.7   38206.9
         1  am_put_12100      38  161.5  EUR     6138.4       -5.7   51365.2
         2  am_put_12150      20  171.3  EUR     3426.8       -3.3   27894.5
         3  am_put_12200      12  183.9  EUR     2206.6       -2.2   18479.7
         4  am_put_12250      37  197.4  EUR     7302.8       -7.3   59423.5
         5  am_put_12300      37  212.3  EUR     7853.9       -8.2   65911.9
         6  am_put_12350      36  228.4  EUR     8224.1       -9.0   70969.4
         7  am_put_12400      16  244.3  EUR     3908.4       -4.3   32871.4
         8  am_put_12450      17  262.7  EUR     4465.6       -5.1   37451.2
```

```
9   am_put_12500     16 283.4  EUR     4534.8       -5.2 36158.2
10  am_put_12550     38 305.3  EUR    11602.3      -13.3 86869.9
11  am_put_12600     10 330.4  EUR     3303.9       -3.9 22144.5
12  am_put_12650     38 355.5  EUR    13508.3      -16.0 89124.8
13  am_put_12700     40 384.2  EUR    15367.5      -18.6 90871.2
14  am_put_12750     13 413.5  EUR     5375.7       -6.5 28626.0
15  am_put_12800     49 445.0  EUR    21806.6      -26.3 105287.3
16  am_put_12850     30 477.4  EUR    14321.8      -17.0 60757.2
17  am_put_12900     33 510.3  EUR    16840.1      -19.7 69163.6
18  am_put_12950     40 544.4  EUR    21777.0      -24.9 80472.3
19  am_put_13000     35 582.3  EUR    20378.9      -22.9 66522.6

In [58]: results[['pos_value','pos_delta','pos_vega']].sum().round(1)
Out[58]: pos_value     197346.2
         pos_delta       -224.0
         pos_vega     1138571.1
         dtype: float64
```

❶ `final_date` 参数后来被重置为组合中所有期权的最后到期日。

❷ 组合中的美式看跌期权标的都是同一个风险因素——DAX 30 股票指数。

❸ 实例化 `dx.derivatives_portfolio` 对象。

因为使用最小二乘蒙特卡洛方法来估算多种美式期权的价值，这种估算的计算特别密集，所以，所有统计量的估算需要花费一点时间。我们只处理美式看跌期权的多头头寸，由此组合的 Delta 值较小，Vega 值较大。

21.4　Python 代码

下面是从 Eikon Data API 读取德国 DAX 30 股票指数期权数据的代码：

```
In [1]: import eikon as ek ❶
        import pandas as pd
        import datetime as dt
        import configparser as cp

In [2]: cfg = cp.ConfigParser() ❷
        cfg.read('eikon.cfg') ❷
Out[2]: ['eikon.cfg']

In [3]: ek.set_app_id(cfg['eikon']['app_id']) ❷

In [4]: fields = ['CF_DATE', 'EXPIR_DATE', 'PUTCALLIND',
                  'STRIKE_PRC', 'CF_CLOSE', 'IMP_VOLT'] ❸
```

```
In [5]: dax = ek.get_data('0#GDAXN8*.EX', fields=fields)[0] ❹

In [6]: dax.info() ❹

        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 115 entries, 0 to 114
        Data columns (total 7 columns):
        Instrument    115 non-null object
        CF_DATE       115 non-null object
        EXPIR_DATE    114 non-null object
        PUTCALLIND    114 non-null object
        STRIKE_PRC    114 non-null float64
        CF_CLOSE      115 non-null float64
        IMP_VOLT      114 non-null float64
        dtypes: float64(3), object(4)
        memory usage: 6.4+ KB


In [7]: dax['Instrument'] = dax['Instrument'].apply(
            lambda x: x.replace('/', '')) ❺

In [8]: dax.set_index('Instrument').head(10)
Out[8]:               CF_DATE EXPIR_DATE PUTCALLIND STRIKE_PRC CF_CLOSE \
        Instrument
        .GDAXI          2018-04-27       None    None      NaN 12500.47
        GDAX105000G8.EX 2018-04-27 2018-07-20    CALL 10500.0  2040.80
        GDAX105000S8.EX 2018-04-27 2018-07-20    PUT  10500.0    32.00
        GDAX108000G8.EX 2018-04-27 2018-07-20    CALL 10800.0  1752.40
        GDAX108000S8.EX 2018-04-26 2018-07-20    PUT  10800.0    43.80
        GDAX110000G8.EX 2018-04-27 2018-07-20    CALL 11000.0  1562.80
        GDAX110000S8.EX 2018-04-27 2018-07-20    PUT  11000.0    54.50
        GDAX111500G8.EX 2018-04-27 2018-07-20    CALL 11150.0  1422.50
        GDAX111500S8.EX 2018-04-27 2018-07-20    PUT  11150.0    64.30
        GDAX112000G8.EX 2018-04-27 2018-07-20    CALL 11200.0  1376.10

                        IMP_VOLT
        Instrument
        .GDAXI               NaN
        GDAX105000G8.EX    23.59
        GDAX105000S8.EX    23.59
        GDAX108000G8.EX    22.02
        GDAX108000S8.EX    22.02
        GDAX110000G8.EX    21.00
        GDAX110000S8.EX    21.00
```

```
          GDAX111500G8.EX    20.24
          GDAX111500S8.EX    20.25
          GDAX112000G8.EX    19.99

In [9]: dax.to_csv('../../source/tr_eikon_option_data.csv') ❻
```

❶ 导入 `eikon` Python 包装器库。

❷ 读取 Eikon Data API 的登录凭据。

❸ 定义读取的数据字段。

❹ 读取 2018 年 7 月到期的期权数据。

❺ 用金融工具名称代替斜杠。

❻ 将数据集写入一个 CSV 文件。

21.5 结语

本章介绍了 DX 分析库更大规模、更实际的用例——估算基于德国 DAX 30 股票指数的非交易美式期权投资组合的价值。本章介绍了任何实际应用中都涉及的 3 个重要任务，具体如下。

数据获取

最新、正确的市场数据是任何衍生品分析中建模与估值工作的基础；我们需要 DAX 30 指数数据以及期权数据。

模型检验

为了以同市场一致的方式估值、管理和对冲非交易期权，我们需要用期权市场报价（在到期日和行权价上相对应）检验模型（模拟对象）参数。我们选择的模型是跳跃扩散，这种方法在某些情况下适合建立股票指数的模型。检验结果相当不错，但是模型仅提供 3 种自由度（lambda 是跳跃密度，mu 是预期跳跃规模，delta 是跳跃规模可变性）。

投资组合估值

根据市场数据和经过检验的模型，我们建立了包含美式 DAX 30 看跌期权投资组合的模型，并生成重要统计量（头寸价值、Delta 和 Vega）。

本章中的实际用例说明了 DX 库的灵活性和能力；它本质上使我们可以对付任何衍生品分析任务。基本方法和架构使这种应用大体上可以与欧式期权 Black-Scholes-Merton 分析公式的基准用例相媲美。一旦定义了估值对象，就可以将与其类似的方式用于某个分析公式——不管表面下隐藏着怎样的事实，都需要大量应用数值化例程和算法。

21.6 延伸阅读

和前几章一样，如下图书是本章介绍主题的很好参考，尤其是期权定价模型的检验：

- Hilpisch, Yves (2015). *Derivatives Analytics with Python*. Chichester, England: Wiley Finance.

关于衍生品投资组合的一致性估值和管理，可参见第 20 章末的资源。

附录 A

日期与时间

正如大部分科学学科一样，日期和时间在金融学中起着重要的作用。本附录介绍 Python 编程中这一主题的不同方面。当然，本附录并不全面。不过，它可以作为支持日期和时间信息建模的 Python 生态系统主要领域的入门介绍。

A.1 Python

Python 标准库中的 datetime 模块实现了非常重要的日期和时间的相关任务：

```
In [1]: from pylab import mpl, plt
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline

In [2]: import datetime as dt

In [3]: dt.datetime.now() ❶
Out[3]: datetime.datetime(2018, 10, 19, 15, 17, 32, 164295)

In [4]: to = dt.datetime.today() ❶
        to
Out[4]: datetime.datetime(2018, 10, 19, 15, 17, 32, 177092)

In [5]: type(to)
Out[5]: datetime.datetime

In [6]: dt.datetime.today().weekday() ❷
Out[6]: 4
```

❶ 返回准确的日期和系统时间。

❷ 返回以数字表示的周日，0 表示星期一。

当然，datetime 对象可以自由定义：

```
In [7]: d = dt.datetime(2020, 10, 31, 10, 5, 30, 500000) ❶
        d
Out[7]: datetime.datetime(2020, 10, 31, 10, 5, 30, 500000)

In [8]: str(d) ❷
Out[8]: '2020-10-31 10:05:30.500000'

In [9]: print(d) ❸
        2020-10-31 10:05:30.500000

In [10]: d.year ❹
Out[10]: 2020

In [11]: d.month ❺
Out[11]: 10

In [12]: d.day ❻
Out[12]: 31

In [13]: d.hour ❼
Out[13]: 10
```

❶ 自定义 datetime 对象。

❷ 字符串表示。

❸ 打印对象。

❹ 对象的 year 属性。

❺ 对象的 month 属性。

❻ 对象的 day 属性。

❼ 对象的 hour 属性。

转换和拆分很容易实现：

```
In [14]: o = d.toordinal() ❶
         o
Out[14]: 737729

In [15]: dt.datetime.fromordinal(o) ❷
Out[15]: datetime.datetime(2020, 10, 31, 0, 0)
```

```
In [16]: t = dt.datetime.time(d) ❸
         t
Out[16]: datetime.time(10, 5, 30, 500000)

In [17]: type(t)
Out[17]: datetime.time

In [18]: dd = dt.datetime.date(d) ❹
         dd
Out[18]: datetime.date(2020, 10, 31)

In [19]: d.replace(second=0, microsecond=0) ❺
Out[19]: datetime.datetime(2020, 10, 31, 10, 5)
```

❶ 转换为序数。

❷ 从序数转换。

❸ 拆分 `time` 部分。

❹ 拆分 `date` 部分。

❺ 将所选值设置为 0。

`timedelta` 是另一类对象，它从 `datetime` 对象的数学运算（即求取两个此类对象的差值）得出：

```
In [20]: td = d - dt.datetime.now() ❶
         td
Out[20]: datetime.timedelta(days=742, seconds=67678, microseconds=169720)

In [21]: type(td) ❷
Out[21]: datetime.timedelta

In [22]: td.days
Out[22]: 742

In [23]: td.seconds
Out[23]: 67678

In [24]: td.microseconds
Out[24]: 169720

In [25]: td.total_seconds() ❸
Out[25]: 64176478.16972
```

❶ 两个 `datetime` 对象的差值。

❷ 产生一个 `timedelta` 对象。

❸ 差值以秒表示。

将 `datetime` 对象转换为不同的表示以及从其他对象（如字符串对象）生成 `datetime` 对象都有多种方式。详情参见 `datetime` 模块的文档。下面是几个例子：

```
In [26]: d.isoformat() ❶
Out[26]: '2020-10-31T10:05:30.500000'

In [27]: d.strftime('%A, %d. %B %Y %I:%M%p') ❷
Out[27]: 'Saturday, 31. October 2020 10:05AM'

In [28]: dt.datetime.strptime('2017-03-31', '%Y-%m-%d') ❸
Out[28]: datetime.datetime(2017, 3, 31, 0, 0)

In [29]: dt.datetime.strptime('30-4-16', '%d-%m-%y') ❸
Out[29]: datetime.datetime(2016, 4, 30, 0, 0)

In [30]: ds = str(d)
         ds
Out[30]: '2020-10-31 10:05:30.500000'

In [31]: dt.datetime.strptime(ds, '%Y-%m-%d %H:%M:%S.%f') ❸
Out[31]: datetime.datetime(2020, 10, 31, 10, 5, 30, 500000)
```

❶ ISO 格式字符串表示。

❷ 字符串表示的模板。

❸ 根据模板从字符串转换的 `datetime` 对象。

除了 `now()` 和 `today()` 函数之外，还有 `utcnow()` 函数，该函数给出 UTC（世界调整时间，过去称作格林尼治标准时间 GMT）形式的日期和时间的准确信息。这一时间和作者所在时区（CET，中欧标准时间）相差 1 个或 2 个小时：

```
In [32]: dt.datetime.now()
Out[32]: datetime.datetime(2018, 10, 19, 15, 17, 32, 438889)

In [33]: dt.datetime.utcnow() ❶
Out[33]: datetime.datetime(2018, 10, 19, 13, 17, 32, 448897)

In [34]: dt.datetime.now() - dt.datetime.utcnow() ❷
Out[34]: datetime.timedelta(seconds=7199, microseconds=999995)
```

❶ 返回当前 UTC 时间。

❷ 返回当地时间和 UTC 时间的差值。

datetime 模块的另一个类是 `tzinfo` 类，这是一个通用时区类，包含 `utcoffset()`、`dst()`和 `tzname()`方法。DST 是夏令时（Daylight Saving Time）的意思。UTC 时间的定义采用如下形式：

```
In [35]: class UTC(dt.tzinfo):
             def utcoffset(self, d):
                 return dt.timedelta(hours=0) ❶
             def dst(self, d):
                 return dt.timedelta(hours=0) ❷
             def tzname(self, d):
                 return 'UTC'

In [36]: u = dt.datetime.utcnow()

In [37]: u
Out[37]: datetime.datetime(2018, 10, 19, 13, 17, 32, 474585)

In [38]: u = u.replace(tzinfo=UTC()) ❷

In [39]: u
Out[39]: datetime.datetime(2018, 10, 19, 13, 17, 32, 474585, tzinfo=
<__main__.UTC
          object at 0x11c9a2320>)

In [40]: class CEST(dt.tzinfo):
             def utcoffset(self, d):
                 return dt.timedelta(hours=2) ❸
             def dst(self, d):
                 return dt.timedelta(hours=1) ❸
             def tzname(self, d):
                 return 'CEST'

In [41]: c = u.astimezone(CEST()) ❹
         c
Out[41]: datetime.datetime(2018, 10, 19, 15, 17, 32, 474585,
          tzinfo=<__main__.CEST object at 0x11c9a2cc0>)

In [42]: c - c.dst() ❺
Out[42]: datetime.datetime(2018, 10, 19, 14, 17, 32, 474585,
          tzinfo=<__main__.CEST object at 0x11c9a2cc0>)
```

❶ UTC 没有时移。

❷ 通过 `replace()`方法连接 `dt.tzinfo` 对象。

❸ CEST 的常规及 DST（夏令时）时移。

❹ 将 UTC 时区转换成 CEST 时区。

❺ 给出转换后 datetime 对象的夏令时时间。

Python 模块 pytz 可以实现全世界非常重要的时区：

```
In [43]: import pytz

In [44]: pytz.country_names['US'] ❶
Out[44]: 'United States'

In [45]: pytz.country_timezones['BE'] ❷
Out[45]: ['Europe/Brussels']

In [46]: pytz.common_timezones[-10:] ❸
Out[46]: ['Pacific/Wake',
          'Pacific/Wallis',
          'US/Alaska',
          'US/Arizona',
          'US/Central',
          'US/Eastern',
          'US/Hawaii',
          'US/Mountain',
          'US/Pacific',
          'UTC']
```

❶ 单一国家。

❷ 单一时区。

❸ 常见的一些时区。

有了 pytz，通常就没有必要自定义 tzinfo 对象了：

```
In [47]: u = dt.datetime.utcnow()

In [48]: u = u.replace(tzinfo=pytz.utc) ❶

In [49]: u
Out[49]: datetime.datetime(2018, 10, 19, 13, 17, 32, 611417, tzinfo=<UTC>)

In [50]: u.astimezone(pytz.timezone('CET')) ❷
Out[50]: datetime.datetime(2018, 10, 19, 15, 17, 32, 611417, tzinfo=<DstTzInfo
          'CET' CEST+2:00:00 DST>)

In [51]: u.astimezone(pytz.timezone('GMT')) ❷
Out[51]: datetime.datetime(2018, 10, 19, 13, 17, 32, 611417, tzinfo=
<StaticTzInfo 'GMT'>)
```

```
In [52]: u.astimezone(pytz.timezone('US/Central')) ❷
Out[52]: datetime.datetime(2018, 10, 19, 8, 17, 32, 611417, tzinfo=<DstTzInfo
          'US/Central' CDT-1 day, 19:00:00 DST>)
```

❶ 通过 pytz 定义 `tzinfo` 对象。

❷ 将 `datetime` 对象转换为不同时区。

A.2 NumPy

NumPy 也提供了处理日期与时间信息的功能：

```
In [53]: import numpy as np

In [54]: nd = np.datetime64('2020-10-31') ❶
         nd
Out[54]: numpy.datetime64('2020-10-31')

In [55]: np.datetime_as_string(nd) ❶
Out[55]: '2020-10-31'

In [56]: np.datetime_data(nd) ❷
Out[56]: ('D', 1)

In [57]: d
Out[57]: datetime.datetime(2020, 10, 31, 10, 5, 30, 500000)

In [58]: nd = np.datetime64(d) ❸
         nd
Out[58]: numpy.datetime64('2020-10-31T10:05:30.500000')

In [59]: nd.astype(dt.datetime) ❹
Out[59]: datetime.datetime(2020, 10, 31, 10, 5, 30, 500000)
```

❶ 从字符串对象和字符串表示中构造。

❷ 关于数据本身的元信息（类型和大小）。

❸ 从 `datetime` 对象中构造。

❹ 转换为 `datetime` 对象。

构造此类对象的另一种方法是提供一个字符串对象，例如，包含年、月及频率信息的字符串。注意，对象值默认为该月的第一天。也可以根据列表对象构造 `ndarray` 对象：

```
In [60]: nd = np.datetime64('2020-10', 'D')
         nd
Out[60]: numpy.datetime64('2020-10-01')

In [61]: np.datetime64('2020-10') == np.datetime64('2020-10-01')
Out[61]: True

In [62]: np.array(['2020-06-10', '2020-07-10', '2020-08-10'], dtype=
'datetime64')
Out[62]: array(['2020-06-10', '2020-07-10', '2020-08-10'], dtype=
'datetime64[D]')

In [63]: np.array(['2020-06-10T12:00:00', '2020-07-10T12:00:00',
                   '2020-08-10T12:00:00'], dtype='datetime64[s]')
Out[63]: array(['2020-06-10T12:00:00', '2020-07-10T12:00:00',
                '2020-08-10T12:00:00'], dtype='datetime64[s]')
```

我们也可以使用`np.arange()`函数生成一个日期范围。很容易使用不同的频率（例如，天、月或者周）：

```
In [64]: np.arange('2020-01-01', '2020-01-04', dtype='datetime64') ❶
Out[64]: array(['2020-01-01', '2020-01-02', '2020-01-03'], dtype=
'datetime64[D]')

In [65]: np.arange('2020-01-01', '2020-10-01', dtype='datetime64[M]') ❷
Out[65]: array(['2020-01', '2020-02', '2020-03', '2020-04', '2020-05',
                '2020-06', '2020-07', '2020-08', '2020-09'],
                dtype='datetime64[M]')

In [66]: np.arange('2020-01-01', '2020-10-01', dtype='datetime64[W]')[:10] ❸
Out[66]: array(['2019-12-26', '2020-01-02', '2020-01-09', '2020-01-16',
                '2020-01-23', '2020-01-30', '2020-02-06', '2020-02-13',
                '2020-02-20', '2020-02-27'], dtype='datetime64[W]')

In [67]: dtl = np.arange('2020-01-01T00:00:00', '2020-01-02T00:00:00',
                         dtype='datetime64[h]') ❹
         dtl[:10]
Out[67]: array(['2020-01-01T00', '2020-01-01T01', '2020-01-01T02',
                '2020-01-01T03', '2020-01-01T04', '2020-01-01T05',
'2020-01-01T06',
                '2020-01-01T07', '2020-01-01T08', '2020-01-01T09'],
                dtype='datetime64[h]')

In [68]: np.arange('2020-01-01T00:00:00', '2020-01-02T00:00:00',
                   dtype='datetime64[s]')[:10] ❺
Out[68]: array(['2020-01-01T00:00:00', '2020-01-01T00:00:01',
```

```
                '2020-01-01T00:00:02', '2020-01-01T00:00:03',
                '2020-01-01T00:00:04', '2020-01-01T00:00:05',
                '2020-01-01T00:00:06', '2020-01-01T00:00:07',
                '2020-01-01T00:00:08', '2020-01-01T00:00:09'],
                dtype='datetime64[s]')

In [69]: np.arange('2020-01-01T00:00:00', '2020-01-02T00:00:00',
                   dtype='datetime64[ms]')[:10] ❻
Out[69]: array(['2020-01-01T00:00:00.000', '2020-01-01T00:00:00.001',
                '2020-01-01T00:00:00.002', '2020-01-01T00:00:00.003',
                '2020-01-01T00:00:00.004', '2020-01-01T00:00:00.005',
                '2020-01-01T00:00:00.006', '2020-01-01T00:00:00.007',
                '2020-01-01T00:00:00.008', '2020-01-01T00:00:00.009'],
                dtype='datetime64[ms]')
```

❶ 每日频率。

❷ 每月频率。

❸ 每周频率。

❹ 每小时频率。

❺ 每秒频率。

❻ 每毫秒频率。

绘制日期-时间和时间序列数据的图表有时候很困难。matplotlib 对标准的 `datetime` 对象有很好的支持。将 `datetime64` 信息转换为 `datetime` 信息通常可以取得成功，正如下面的例子，其结果如图 A-1 所示：

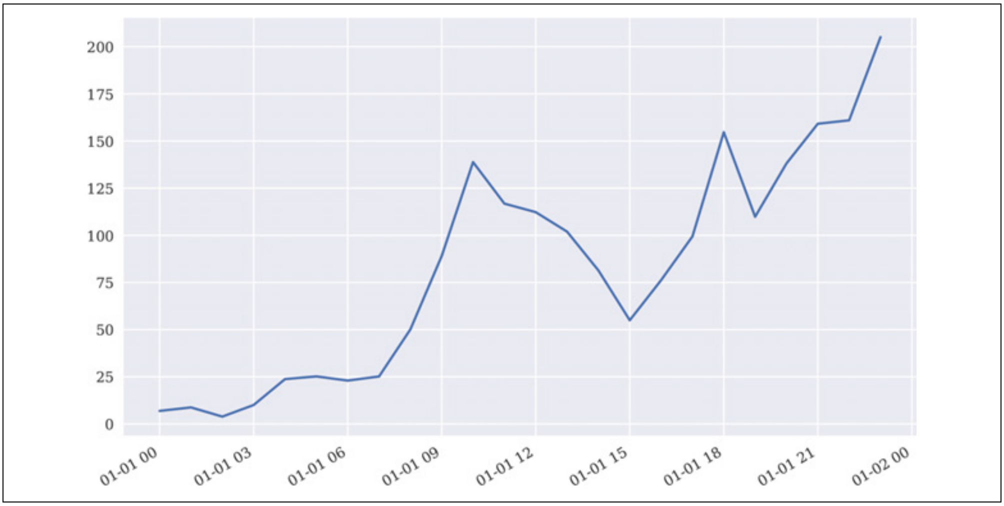


图 A-1　图上的 *x* 轴的 **`datetime`** 刻度自动格式化

```
In [70]: import matplotlib.pyplot as plt
         %matplotlib inline

In [71]: np.random.seed(3000)
         rnd = np.random.standard_normal(len(dtl)).cumsum() ** 2

In [72]: fig = plt.figure(figsize=(10, 6))
         plt.plot(dtl.astype(dt.datetime), rnd) ❶
         fig.autofmt_xdate(); ❷
```

❶ 使用 `datetime` 信息作为 x 值。

❷ 自动格式化 *x* 轴上的 `datetime` 刻度。

A.3 pandas

pandas 库在设计时至少在某种程度上考虑了时间序列数据。因此，该库提供了可以高效处理日期-时间信息的类，如用于时间索引的 `DatetimeIndex` 类。

pandas 引入了 `Timestamp` 对象，它会进一步代替 `datetime` 和 `datetime64` 对象：

```
In [73]: import pandas as pd

In [74]: ts = pd.Timestamp('2020-06-30') ❶
         ts
Out[74]: Timestamp('2020-06-30 00:00:00')

In [75]: d = ts.to_pydatetime() ❷
         d
Out[75]: datetime.datetime(2020, 6, 30, 0, 0)

In [76]: pd.Timestamp(d) ❸
Out[76]: Timestamp('2020-06-30 00:00:00')

In [77]: pd.Timestamp(nd) ❹
Out[77]: Timestamp('2020-10-01 00:00:00')
```

❶ 来自字符串对象的 `Timestamp` 对象。

❷ 来自 `Timestamp` 对象的 `datetime` 对象。

❸ 来自 `datetime` 对象的 `Timestamp` 对象。

❹ 来自 `datetime64` 对象的 `Timestamp` 对象。

另一个重要的类是 `DatetimeIndex` 类，它是 `Timestamp` 对象和一些强大的附加方法的集合。`DatetimeIndex` 对象可以用 `pd.date_range()`函数实例化，该函数在构造

时间索引方面相当灵活而强大（关于该函数的更多细节参见第 8 章）。类型转换是可能的：

```
In [78]: dti = pd.date_range('2020/01/01', freq='M', periods=12) ❶
         dti
Out[78]: DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',
                        '2020-05-31', '2020-06-30', '2020-07-31', '2020-08-31',
                        '2020-09-30', '2020-10-31', '2020-11-30', '2020-12-31'],
                        dtype='datetime64[ns]', freq='M')

In [79]: dti[6]
Out[79]: Timestamp('2020-07-31 00:00:00', freq='M')

In [80]: pdi = dti.to_pydatetime() ❷
         pdi
Out[80]: array([datetime.datetime(2020, 1, 31, 0, 0),
                datetime.datetime(2020, 2, 29, 0, 0),
                datetime.datetime(2020, 3, 31, 0, 0),
                datetime.datetime(2020, 4, 30, 0, 0),
                datetime.datetime(2020, 5, 31, 0, 0),
                datetime.datetime(2020, 6, 30, 0, 0),
                datetime.datetime(2020, 7, 31, 0, 0),
                datetime.datetime(2020, 8, 31, 0, 0),
                datetime.datetime(2020, 9, 30, 0, 0),
                datetime.datetime(2020, 10, 31, 0, 0),
                datetime.datetime(2020, 11, 30, 0, 0),
                datetime.datetime(2020, 12, 31, 0, 0)], dtype=object)

In [81]: pd.DatetimeIndex(pdi) ❸
Out[81]: DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',
                        '2020-05-31', '2020-06-30', '2020-07-31', '2020-08-31',
                        '2020-09-30', '2020-10-31', '2020-11-30', '2020-12-31'],
                        dtype='datetime64[ns]', freq=None)

In [82]: pd.DatetimeIndex(dtl) ❹
Out[82]: DatetimeIndex(['2020-01-01 00:00:00', '2020-01-01 01:00:00',
                        '2020-01-01 02:00:00', '2020-01-01 03:00:00',
                        '2020-01-01 04:00:00', '2020-01-01 05:00:00',
                        '2020-01-01 06:00:00', '2020-01-01 07:00:00',
                        '2020-01-01 08:00:00', '2020-01-01 09:00:00',
                        '2020-01-01 10:00:00', '2020-01-01 11:00:00',
                        '2020-01-01 12:00:00', '2020-01-01 13:00:00',
                        '2020-01-01 14:00:00', '2020-01-01 15:00:00',
                        '2020-01-01 16:00:00', '2020-01-01 17:00:00',
                        '2020-01-01 18:00:00', '2020-01-01 19:00:00',
                        '2020-01-01 20:00:00', '2020-01-01 21:00:00',
```

```
                        '2020-01-01 22:00:00', '2020-01-01 23:00:00'],
                        dtype='datetime64[ns]', freq=None)
```

❶ DatetimeIndex 对象（频率为每月一次，共 12 期）。

❷ DatetimeIndex 对象转换为包含 datetime 对象的 ndarray 对象。

❸ 由包含 datetime 对象的 ndarray 对象转换而来的 DatetimeIndex 对象。

❹ 由包含 datetime64 对象的 ndarray 对象转换而来的 DatetimeIndex 对象。

pandas 可以绘制日期-时间信息的对应图表（参见图 A-2 和第 8 章）:

```
In [83]: rnd = np.random.standard_normal(len(dti)).cumsum() ** 2

In [84]: df = pd.DataFrame(rnd, columns=['data'], index=dti)

In [85]: df.plot(figsize=(10, 6));
```



图 A-2　pandas 图表，*x* 轴的 Timestamp 刻度自动格式化

pandas 还可以很好地与 pytz 模块集成来管理时区：

```
In [86]: pd.date_range('2020/01/01', freq='M', periods=12,
                       tz=pytz.timezone('CET'))
Out[86]: DatetimeIndex(['2020-01-31 00:00:00+01:00', '2020-02-29
          00:00:00+01:00',
         '2020-03-31 00:00:00+02:00', '2020-04-30 00:00:00+02:00',
         '2020-05-31 00:00:00+02:00', '2020-06-30 00:00:00+02:00',
```

```
          '2020-07-31 00:00:00+02:00', '2020-08-31 00:00:00+02:00',
          '2020-09-30 00:00:00+02:00', '2020-10-31 00:00:00+01:00',
          '2020-11-30 00:00:00+01:00', '2020-12-31 00:00:00+01:00'],
                       dtype='datetime64[ns, CET]', freq='M')

In [87]: dti = pd.date_range('2020/01/01', freq='M', periods=12,
tz='US/Eastern')
         dti
Out[87]: DatetimeIndex(['2020-01-31 00:00:00-05:00', '2020-02-29
          00:00:00-05:00',
          '2020-03-31 00:00:00-04:00', '2020-04-30 00:00:00-04:00',
          '2020-05-31 00:00:00-04:00', '2020-06-30 00:00:00-04:00',
          '2020-07-31 00:00:00-04:00', '2020-08-31 00:00:00-04:00',
          '2020-09-30 00:00:00-04:00', '2020-10-31 00:00:00-04:00',
          '2020-11-30 00:00:00-05:00', '2020-12-31 00:00:00-05:00'],
                       dtype='datetime64[ns, US/Eastern]', freq='M')

In [88]: dti.tz_convert('GMT')
Out[88]: DatetimeIndex(['2020-01-31 05:00:00+00:00', '2020-02-29
          05:00:00+00:00',
          '2020-03-31 04:00:00+00:00', '2020-04-30 04:00:00+00:00',
          '2020-05-31 04:00:00+00:00', '2020-06-30 04:00:00+00:00',
          '2020-07-31 04:00:00+00:00', '2020-08-31 04:00:00+00:00',
          '2020-09-30 04:00:00+00:00', '2020-10-31 04:00:00+00:00',
          '2020-11-30 05:00:00+00:00', '2020-12-31 05:00:00+00:00'],
                       dtype='datetime64[ns, GMT]', freq='M')
```

附录 B

BSM 期权类

B.1　类定义

下面是 Black-Scholes-Merton（1973）模型（参见第 3 章，特别是例 3-1）下的欧洲看涨期权类定义。这个基于类的实现可以代替 12.5 节中介绍的基于函数实现：

```
#
# Valuation of European call options in Black-Scholes-Merton model
# incl. vega function and implied volatility estimation
# -- class-based implementation
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
from math import log, sqrt, exp
from scipy import stats


class bsm_call_option(object):
    ''' Class for European call options in BSM model.

    Attributes
    ==========
    S0: float
        initial stock/index level
    K: float
        strike price
    T: float
        maturity (in year fractions)
    r: float
```

```
            constant risk-free short rate
        sigma: float
            volatility factor in diffusion term
        Methods
        =======
        value: float
            returns the present value of call option
        vega: float
            returns the vega of call option
        imp_vol: float
            returns the implied volatility given option quote
        '''

        def __init__(self, S0, K, T, r, sigma):
            self.S0 = float(S0)
            self.K = K
            self.T = T
            self.r = r
            self.sigma = sigma

        def value(self):
            ''' Returns option value.
            '''
            d1 = ((log(self.S0 / self.K) +
                   (self.r + 0.5 * self.sigma ** 2) * self.T) /
                   (self.sigma * sqrt(self.T)))
            d2 = ((log(self.S0 / self.K) +
                   (self.r - 0.5 * self.sigma ** 2) * self.T) /
                   (self.sigma * sqrt(self.T)))
            value = (self.S0 * stats.norm.cdf(d1, 0.0, 1.0) -
                     self.K * exp(-self.r * self.T) * stats.norm.cdf(d2,
0.0, 1.0))
            return value

        def vega(self):
            ''' Returns vega of option.
            '''
            d1 = ((log(self.S0 / self.K) +
                   (self.r + 0.5 * self.sigma ** 2) * self.T) /
                  (self.sigma * sqrt(self.T)))
            vega = self.S0 * stats.norm.pdf(d1, 0.0, 1.0) * sqrt(self.T)
            return vega

        def imp_vol(self, C0, sigma_est=0.2, it=100):
            ''' Returns implied volatility given option price.
```

```
        '''
        option = bsm_call_option(self.S0, self.K, self.T, self.r, sigma_est)
        for i in range(it):
            option.sigma -= (option.value() - C0) / option.vega()
        return option.sigma
```

B.2 类的使用

这个类可以用于如下的交互式 Jupyter Notebook 会话：

```
In [1]: from bsm_option_class import *

In [2]: o = bsm_call_option(100., 105., 1.0, 0.05, 0.2)
        type(o)
Out[2]: bsm_option_class.bsm_call_option

In [3]: value = o.value()
        value
Out[3]: 8.021352235143176

In [4]: o.vega()
Out[4]: 39.67052380842653

In [5]: o.imp_vol(C0=value)
Out[5]: 0.2
```

这个期权类也可用于可视化，例如，不同行权价和到期日的期权价值及 Vega 值，这是拥有期权类的主要好处之一。下面的 Python 代码生成不同到期日-行权价组合的期权统计量：

```
In [6]: import numpy as np
        maturities = np.linspace(0.05, 2.0, 20)
        strikes = np.linspace(80, 120, 20)
        T, K = np.meshgrid(strikes, maturities)
        C = np.zeros_like(K)
        V = np.zeros_like(C)
        for t in enumerate(maturities):
            for k in enumerate(strikes):
                o.T = t[1]
                o.K = k[1]
                C[t[0], k[0]] = o.value()
                V[t[0], k[0]] = o.vega()
```

首先，我们来观察期权的价值。图 B-1 展示了欧洲看涨期权的价值曲面：

```
In [7]: from pylab import cm, mpl, plt
        from mpl_toolkits.mplot3d import Axes3D
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline

In [8]: fig = plt.figure(figsize=(12, 7))
        ax = fig.gca(projection='3d')
        surf = ax.plot_surface(T, K, C, rstride=1, cstride=1,
                    cmap=cm.coolwarm, linewidth=0.5, antialiased=True)
        ax.set_xlabel('strike')
        x.set_ylabel('maturity')
        ax.set_zlabel('European call option value')
        fig.colorbar(surf, shrink=0.5, aspect=5);
```

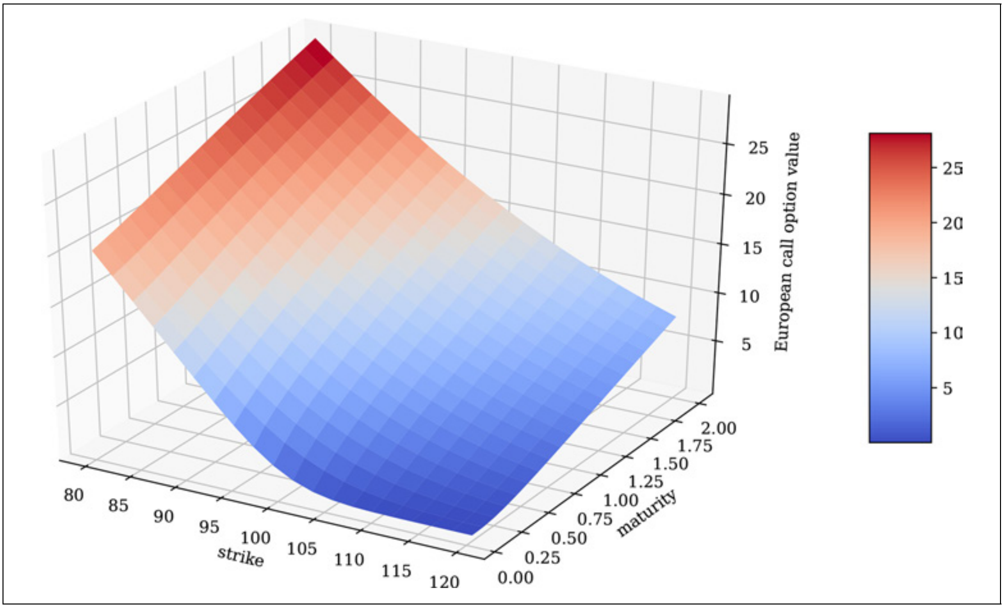


图 B-1　欧式看涨期权价值曲面

其次，观察 Vega 值，图 B-2 展示了欧洲看涨期权的 Vega 曲面：

```
In [9]: fig = plt.figure(figsize=(12, 7))
        ax = fig.gca(projection='3d')
        surf = ax.plot_surface(T, K, V, rstride=1, cstride=1,
                    cmap=cm.coolwarm, linewidth=0.5, antialiased=True)
        ax.set_xlabel('strike')
        ax.set_ylabel('maturity')
        ax.set_zlabel('Vega of European call option')
        fig.colorbar(surf, shrink=0.5, aspect=5);
```

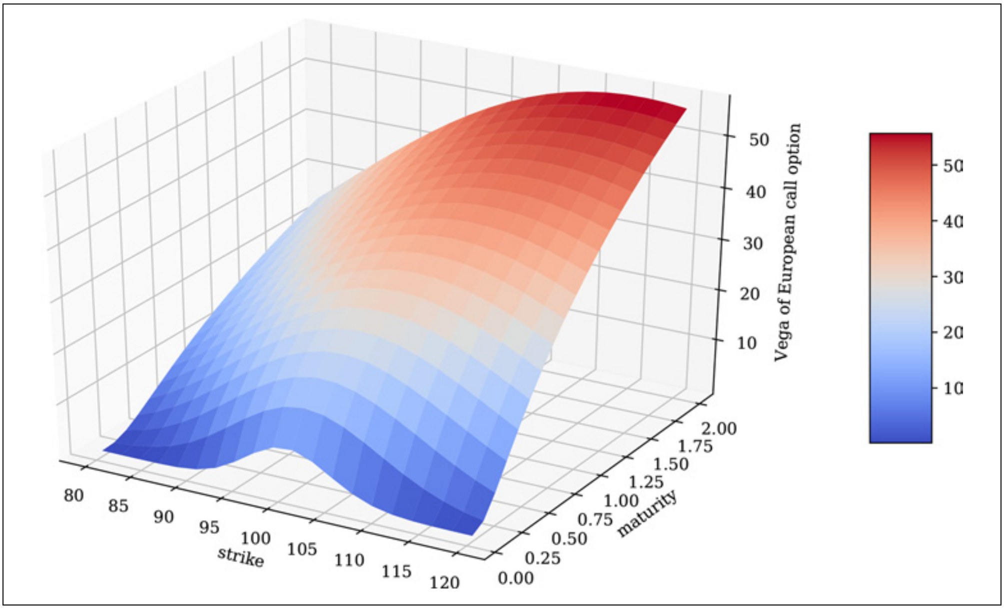


图 B-2　欧洲看涨期权的 Vega 曲面

作者介绍

Yves J. Hilpisch 博士是 Python Quants 集团的创始人和管理合伙人，该集团致力于应用开源技术来解决金融数据科学、人工智能、算法交易和计算金融学等问题。他还是 AI Machine 公司的 CEO，这个公司的专业领域是通过专属策略执行平台来利用人工智能的威力。他还是另外两本图书的作者：

- *Derivatives Analytics with Python* (Wiley, 2015)
- *Listed Volatility and Variance Derivatives* (Wiley, 2017)

Yves 在 CQF 项目讲授计算金融学课程，在 EPAT 项目讲授算法交易。他还是 Python 算法交易大学认证的首个在线培训项目的主管。

Yves 编写了 DX Analytics 金融分析库，并在伦敦、柏林、巴黎和纽约等地组织了关于 Python 计量金融学相关的聚会、会议和训练营。他曾在美国、欧洲和亚洲的技术会议上担任过主讲人。

封面说明

本书封面上的动物是伊斯帕尼奥拉长吻猬，这是一种生活于加勒比海伊斯帕尼奥拉岛（包含海地和多米尼加共和国）的濒危哺乳动物，在海地特别稀少，而在多米尼加共和国中稍微常见一些。

长吻猬以节肢动物、蠕虫、蜗牛和爬虫为食，偶尔也食用根、果实和叶子。长吻猬的重量为 1～2 磅（1 磅 = 0.4536 千克），头和身体长 1 英尺，外加 10 英尺（1 英尺 = 12 英寸 = 0.3048 米）长的尾巴（略有误差）。这种古老的哺乳动物看上去像大的地鼠，多毛，上面为红褐色，下面较浅一些，而尾部、腿和突出的鼻子上则毛发较少。

长吻猬习性相当安静，往往游离于别人的视线之外。当它出门时，行动很笨拙，奔跑时常常绊倒。不过，作为一种夜间型物种，它有敏锐的听觉、嗅觉和触觉，据说，它的独特气味像“小山羊”。

它的下门牙槽分泌有毒的唾液，可以让无脊椎动物瘫痪。长吻猬本身是少数有毒的哺乳动物之一。有时候，在相互争斗时会释放毒气，对长吻猬本身也可能致命。在最初的冲突之后，它们往往确立统治关系，并在同一块区域生活。家族可能长时间居住在一起，它们只在洗澡时饮水。

O’Reilly 图书封面上的许多动物都处于濒危状态；它们对世界很重要。要了解如何帮助它们，可以访问 O’Reilly 官网。

O'REILLY®

Python金融大数据分析

Python已成为数据驱动、AI、金融优先选择的编程语言。现在，一些大型的投资银行和对冲基金均使用Python及其相关技术来构建核心交易与风险管理系统。在本书中，作者向开发人员和量化分析人员介绍了使用Python程序库与工具，完成金融数据科学、算法交易和计算金融任务的方法。

本书的多数代码已针对Python 3做了更新，为读者提供了书中使用的几乎所有示例的可执行、交互版本。本书包括以下5个方面的内容，你可以从中了解Python及其相关技术是如何为金融公司及从业者提供技术框架的。

- **Python和金融：Python交互式金融分析与程序开发入门。**
- **基础知识：学习Python数据类型与结构、NumPy、pandas及其DataFrame类、面向对象编程。**
- **金融数据科学：探索用于金融时间序列数据、I/O操作、推断统计学和机器学习的Python技术与程序库。**
- **算法交易：使用Python来验证和部署自动算法交易策略。**
- **衍生品分析：开发灵活且强大的Python期权、衍生品定价和风险管理程序库。**

"Python因其易于理解的语法、与C/C++的轻松集成和多种数值计算工具，成为了金融分析的极佳选择。它正在快速替代主流金融机构使用的语言与工具，并成为事实上的标准。"

——Kirat Singh

Beacon Platform公司
联合创始人兼CEO

Yves Hilpisch博士是Python Quants集团的创始人和管理合伙人。该集团致力于应用开源技术来解决金融数据科学、人工智能、算法交易和计算金融学等问题。他还是AI Machine公司的创始人和CEO，该公司的主营业务是通过专属策略执行平台来发挥人工智能的威力。他还是Python算法交易大学认证的在线培训项目的主管。

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

封面设计：Karen Montgomery 张健

O'Reilly Media, Inc.授权人民邮电出版社出版

分类建议：计算机／程序设计／Python
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-52133-0

9 787115 521330 >